

Artificial Intelligence and Machine Learning Applications

Prof. Gheith Abandah

Computer Engineering, The University of Jordan

Dec 24, 2020

University of Bahrain, College of Engineering, Postgraduate Studies Forum

Outline

- Introduction to Artificial Intelligence and Machine Learning
- Achievements of Contemporary Artificial Intelligence
- Limitations of Contemporary Artificial Intelligence
- AI Future

Introduction

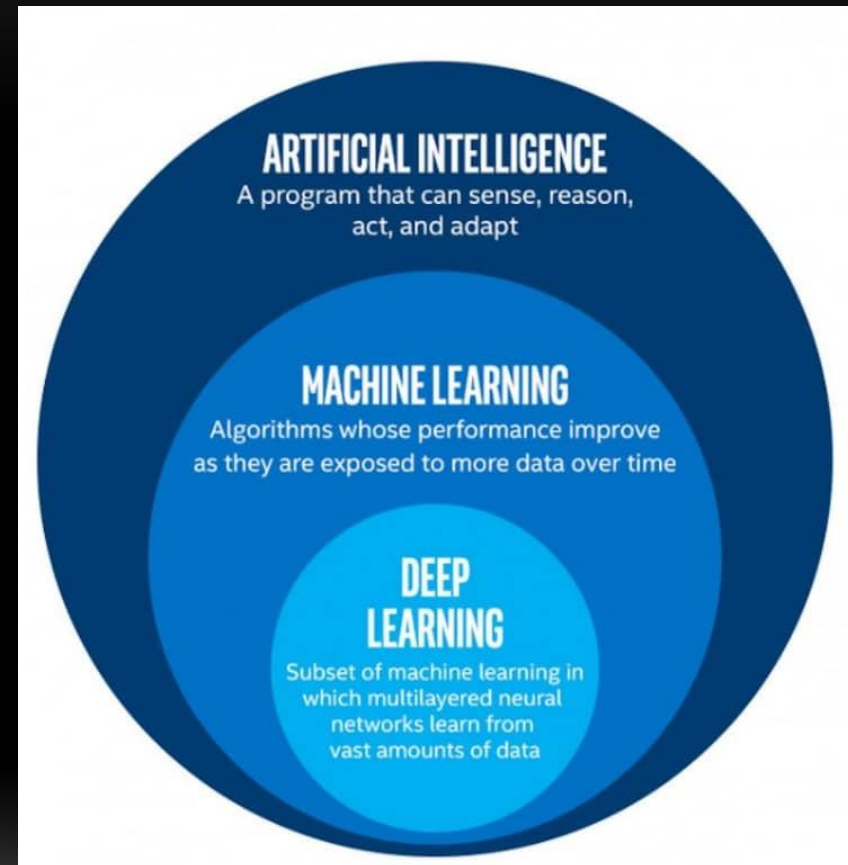
- **Intelligence** Ability to accomplish complex goals
- **Artificial Intelligence (AI)** Non-biological intelligence
- **Narrow Intelligence** Ability to accomplish a narrow set of goals, e.g., play chess or drive a car

Introduction

- **General Intelligence** Ability to accomplish virtually any goal, including learning
- Many large companies and researchers are currently investigating developing General AI
- **Artificial Super Intelligence (ASI)** General Intelligence far beyond human level

Introduction

- **Machine Learning (ML)**
Algorithms whose performance improve as they are exposed to more data
- **Deep Learning (DL)**
Subset of ML using multi-layer neural networks that learn from huge data



Introduction

- **Machine Learning Types**
 - 1. Supervised Learning**
 - 2. Unsupervised Learning**
 - 3. Reinforcement Learning**

Outline

- Introduction to Artificial Intelligence and Machine Learning
- Achievements of Contemporary Artificial Intelligence
- Limitations of Contemporary Artificial Intelligence
- AI Future

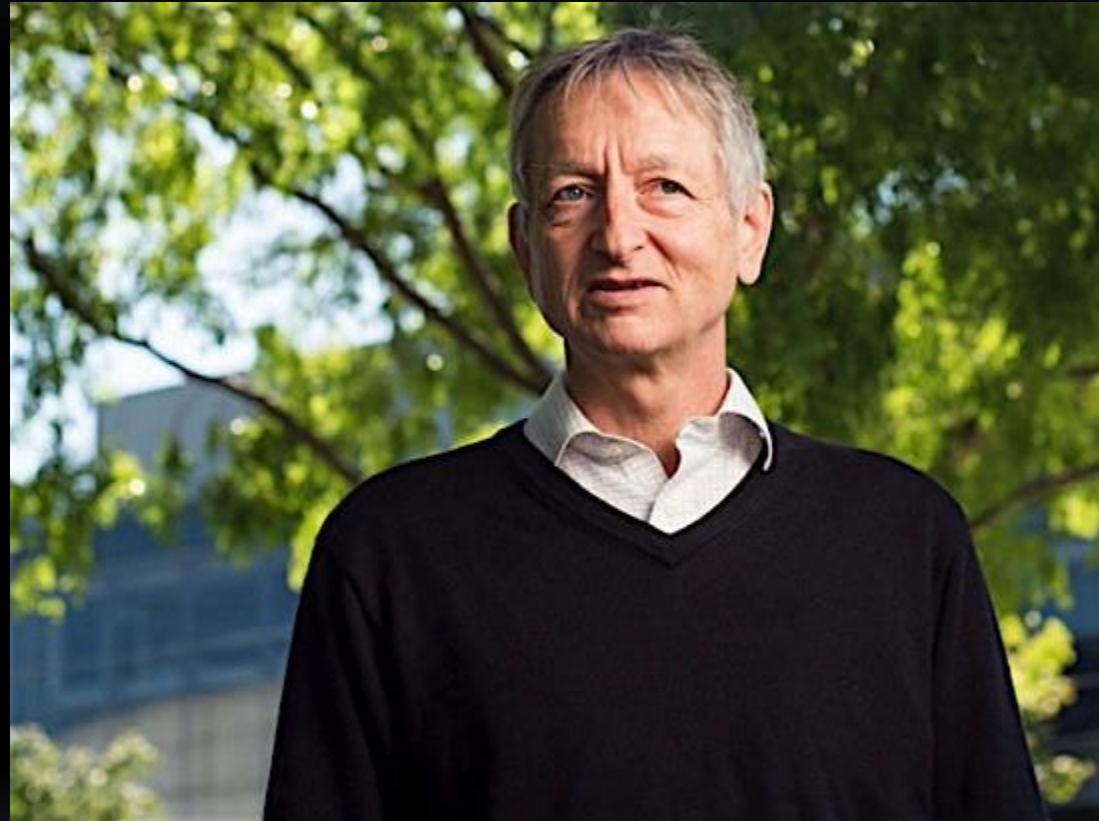
Achievements of Contemporary AI

- Important AI Milestones

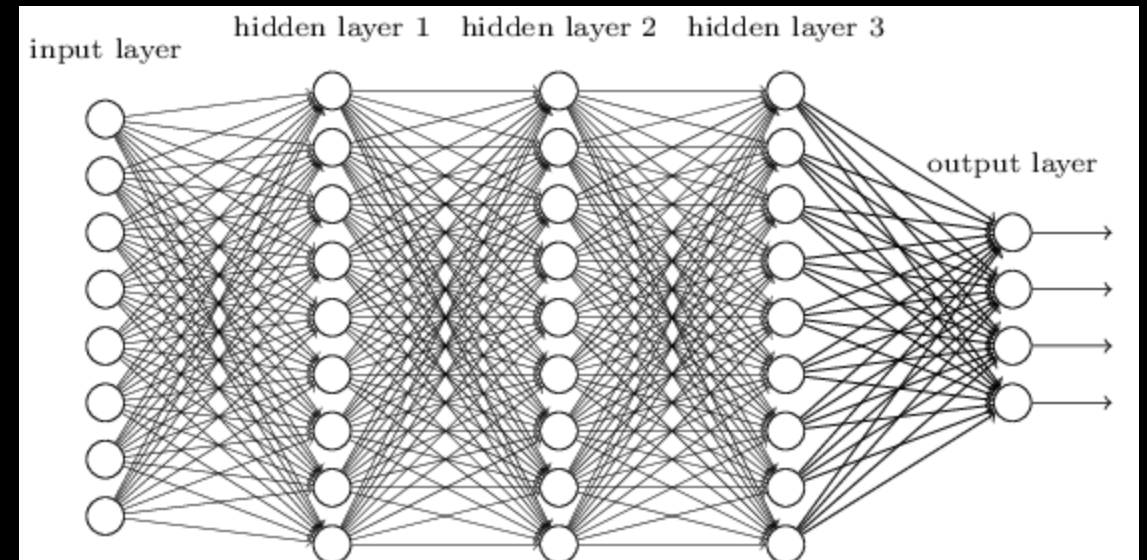
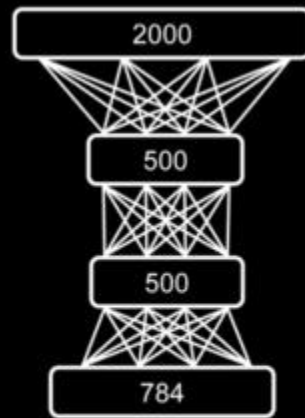
1997: IBM Deep Blue Beats Kasparov



2006: Hinton et al. Train a Deep Neural Network



2006: Hinton et al. Train a Deep Neural Network

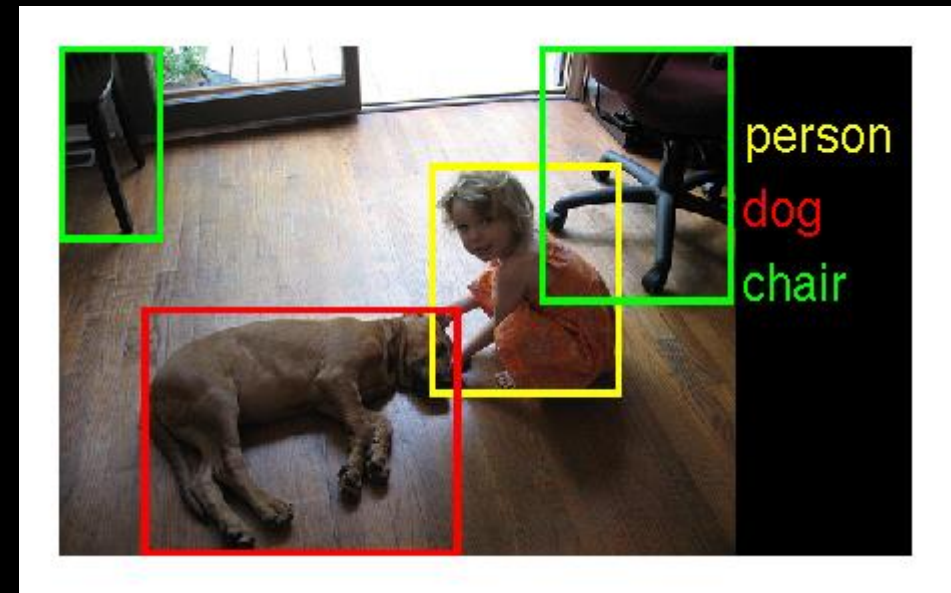


2011: IBM Watson Wins Jeopardy!

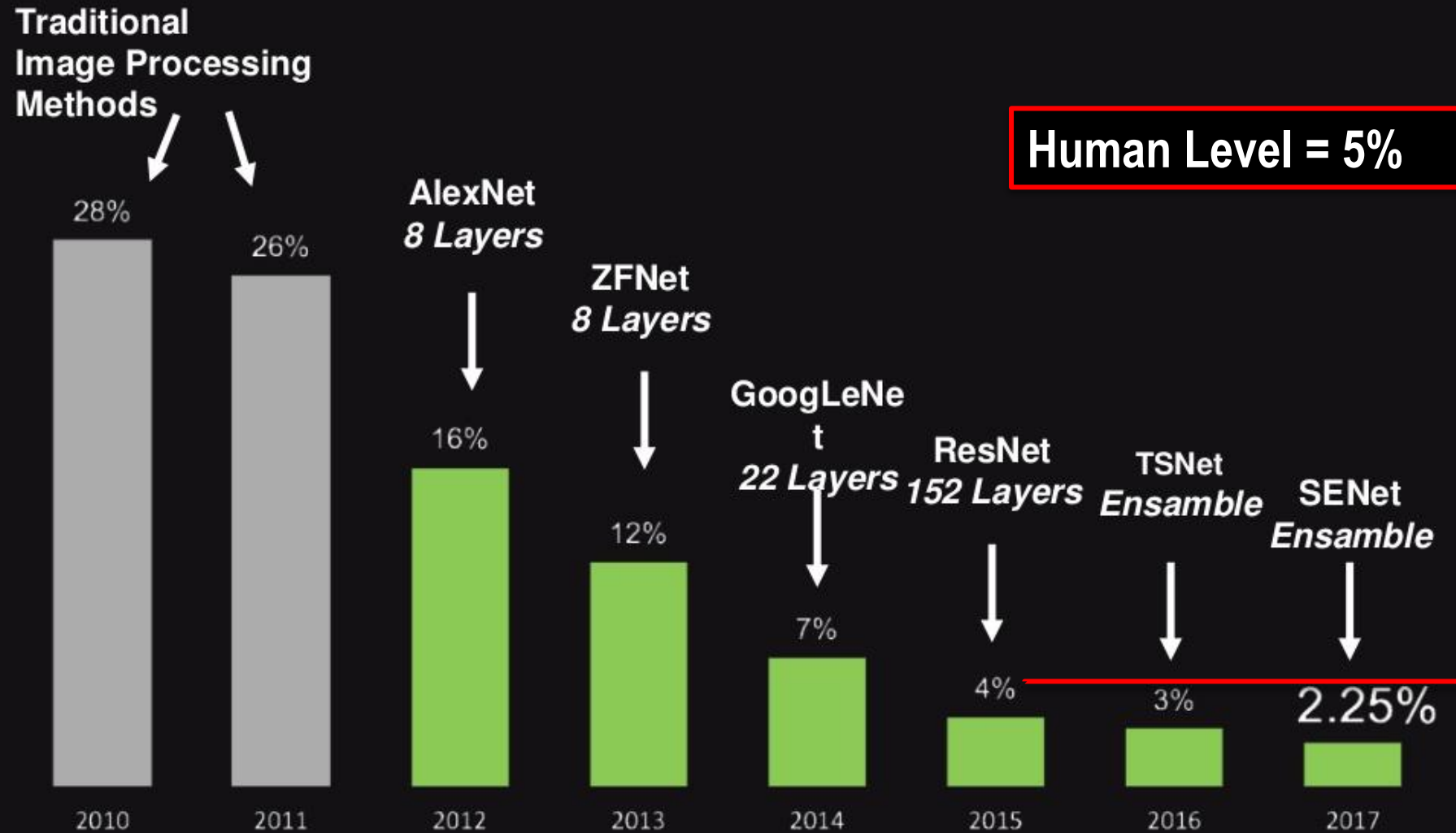


2015: DL Beats Humans in ImageNet

- Large scale visual recognition challenge
 - 1000 classes
 - 1.2 million images



ImageNet Top 5 Error Rate



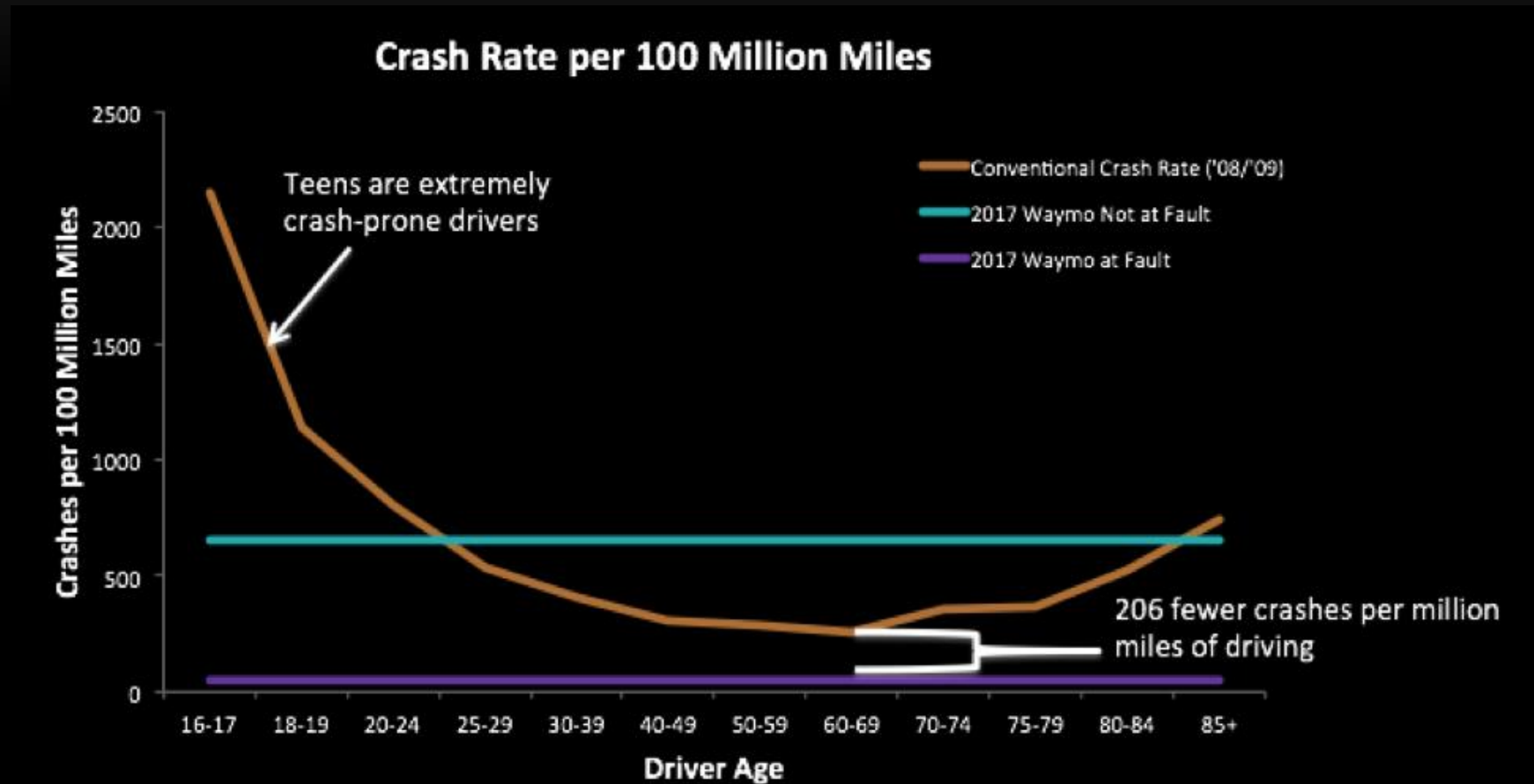
2016: DeepMind AlphaGo Beats Sedol



2017: Google Waymo Reaches Full Self-Driving Capability



Autonomous Vehicles are Saver

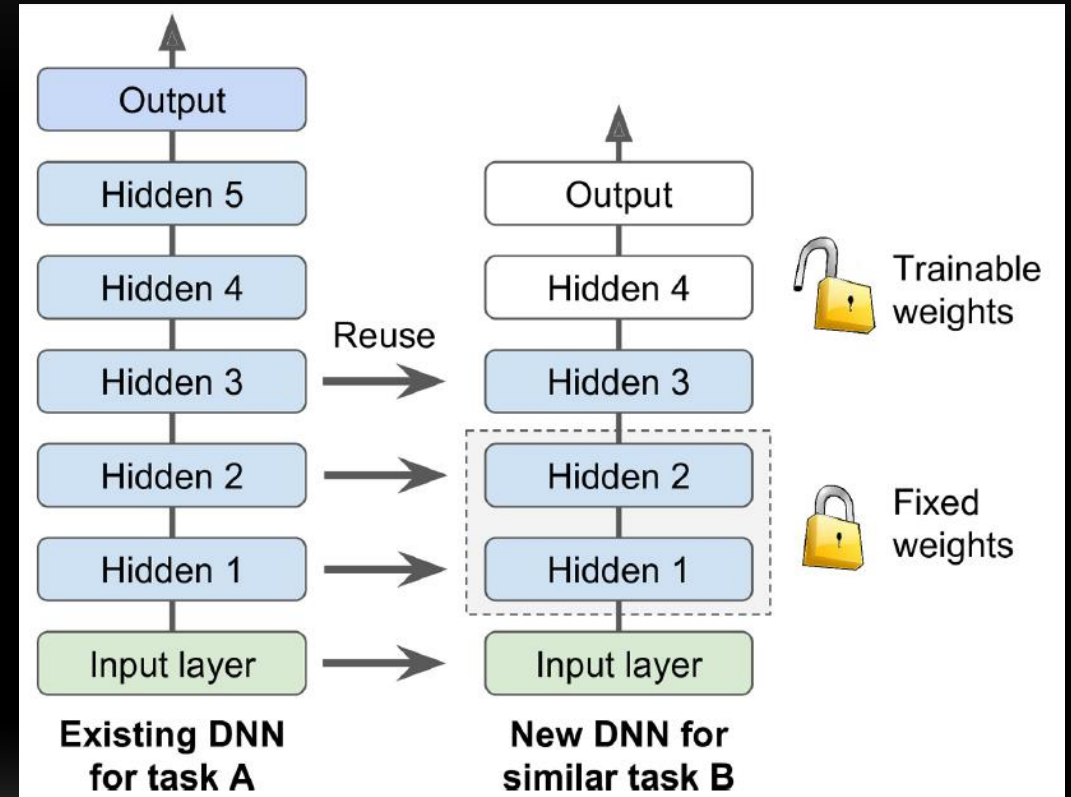


Achievements of Contemporary AI

- 1. Recognizes our voices and photos**
- 2. Recommends who to friend and what to watch and read**
- 3. Helps us in searching and retrieving information**
- 4. Translates natural languages**
- 5. Drives vehicles**
- 6. Secures our cities, systems and detects violations**

Achievements of Contemporary AI

7. Provides cheaper solutions with acceptable qualities
8. Provides trained models we can download and use
9. Allows transfer learning where a model trained for one task can be retrained to solve a different similar task



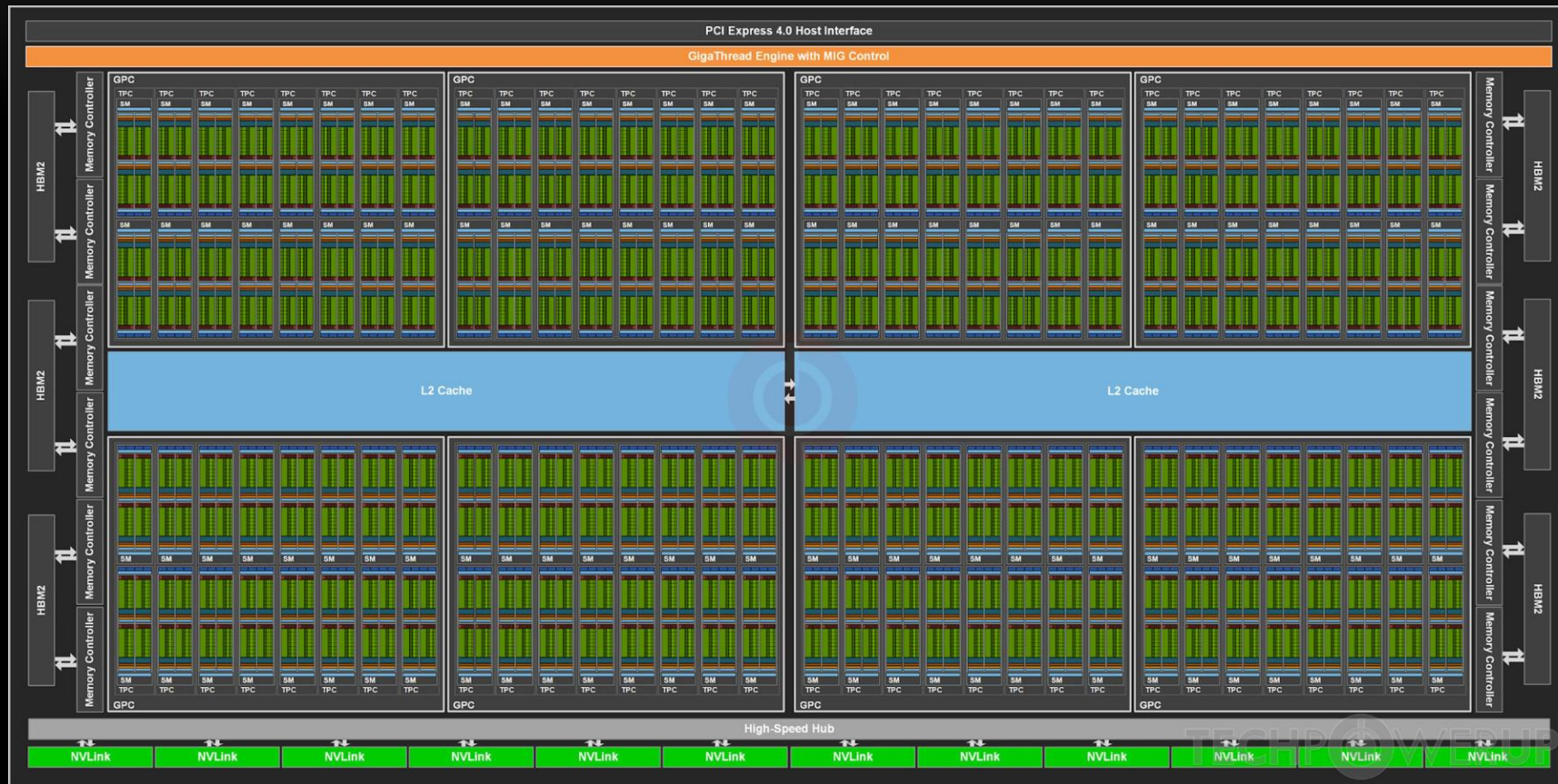
Outline

- Introduction to Artificial Intelligence and Machine Learning
- Achievements of Contemporary Artificial Intelligence
- Limitations of Contemporary Artificial Intelligence
- AI Future

Limitations of Contemporary AI

1. Contemporary AI is narrow AI
2. Deep learning requires huge datasets
3. Deep learning takes long training times
4. Deep learning needs powerful processors and computation accelerators

Nvidia GA100 GPU: 826 mm² chip, 54 billion transistors, 108 SM, 6,912 FP32 CUDA cores, 40 GB memory



Outline

- Introduction to Artificial Intelligence and Machine Learning
- Achievements of Contemporary Artificial Intelligence
- Limitations of Contemporary Artificial Intelligence
- **AI Future**

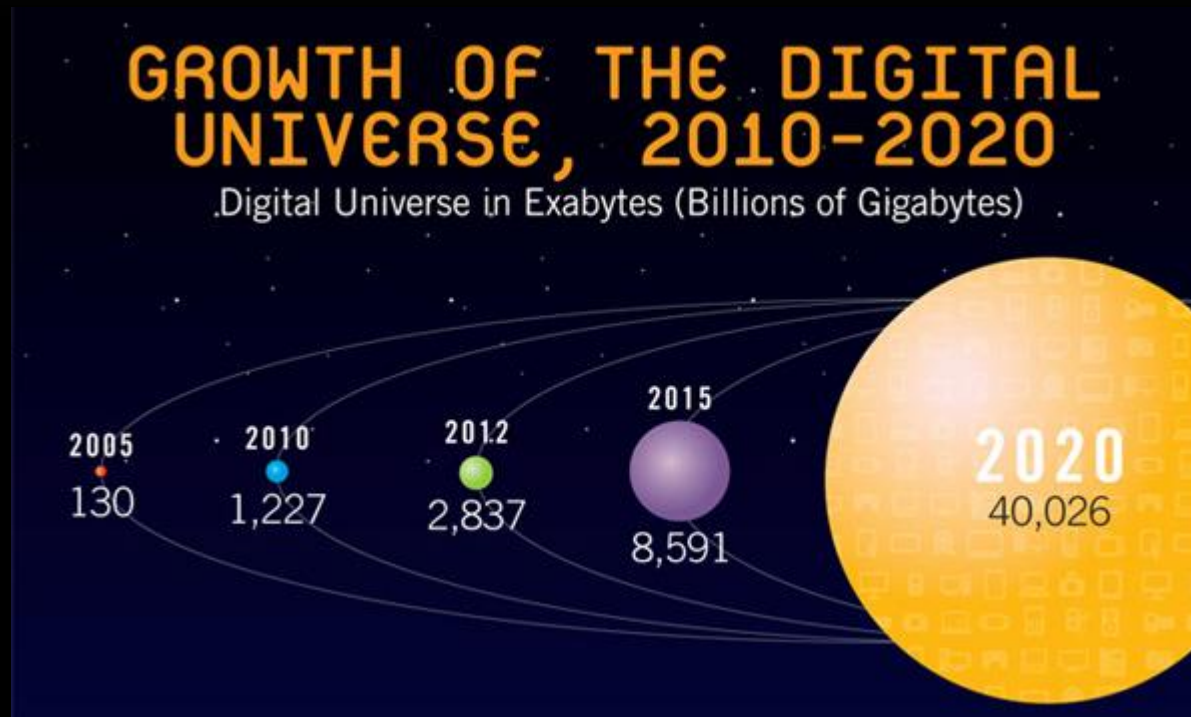
Why AI is Succeeding Now?

1. Data Availability
2. Improved ML Algorithms
3. Fast Processors

AI Will Continue to Succeed

1. More data will be available for machine learning

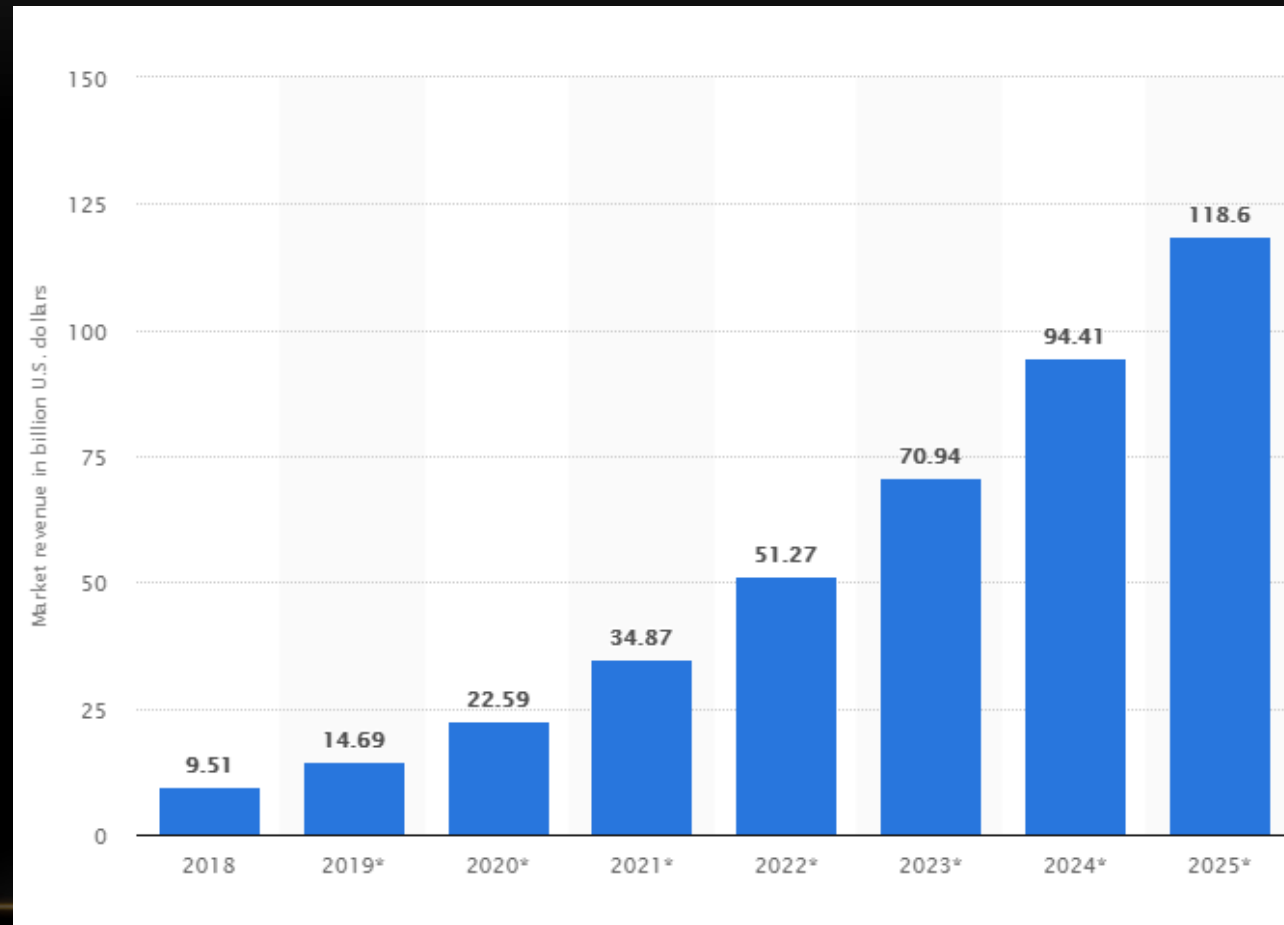
Digital Content Doubles Every Two Years



AI Will Continue to Succeed

1. More data will be available for machine learning
2. Better algorithms and AI applications will continue to develop

Global AI Software Market

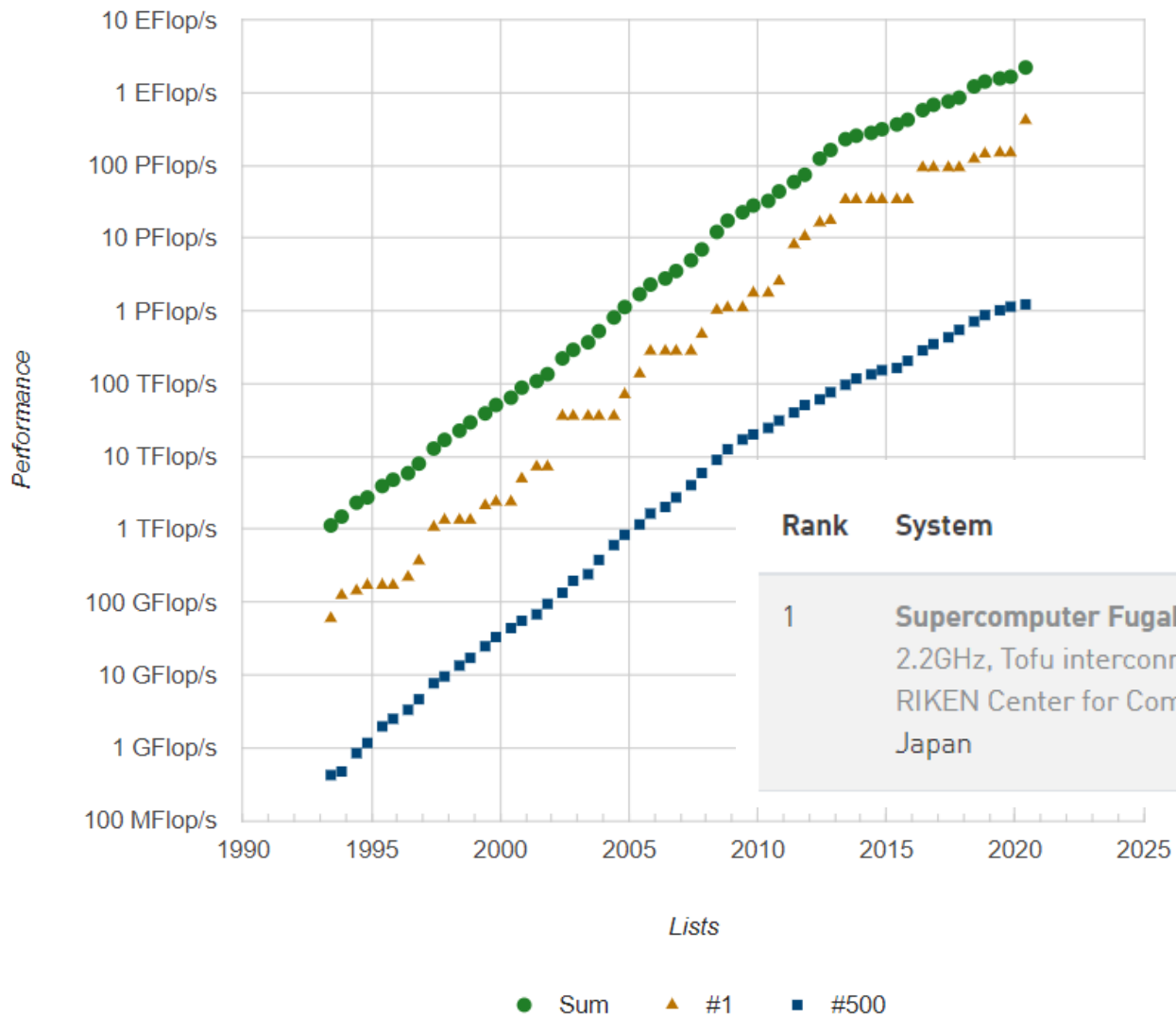


AI Will Continue to Succeed

- 1. More data will be available for machine learning**
- 2. Better algorithms and AI applications will continue to develop**
- 3. Computers will continue to get faster**

Perf. Improves 100x every 10 years

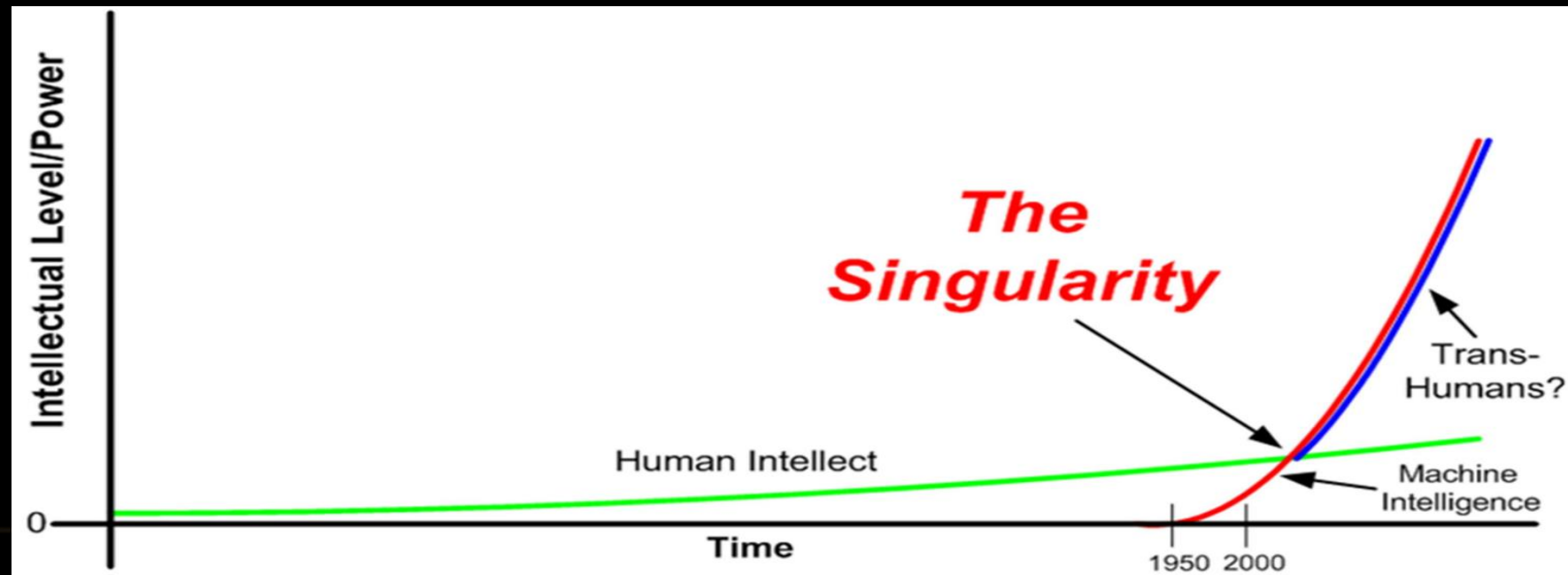
Performance Development



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335

To where we are heading?

- Continued AI development will lead to Singularity



Summary

- Introduction to Artificial Intelligence and Machine Learning
- Achievements of Contemporary Artificial Intelligence
- Limitations of Contemporary Artificial Intelligence
- AI Future

Thank You

- **Email** abandah@ju.edu.jo
- **Facebook** [gheith.abandah](https://www.facebook.com/gheith.abandah)
- **Twitter** [@abandah](https://twitter.com/abandah)
- **LinkedIn** [gheith-abandah](https://www.linkedin.com/company/gheith-abandah)
- **Website** <http://www.abandah.com/gheith>

Introduction to Artificial Intelligence (AI)

Prof. Gheith Abandah

Reference

- Chapter 1: **Introduction to AI**



- Prateek Joshi, **Artificial Intelligence with Python**, Packt, 2017
 - Material: <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python>

Outline

- What is AI?
- Why do we need to study AI?
- Applications of AI
- Branches of AI
- Defining intelligence using Turing Test
- Making machines think like humans
- Building rational agents
- General problem solver
- Building an intelligent agent
- Summary

What is Artificial Intelligence?

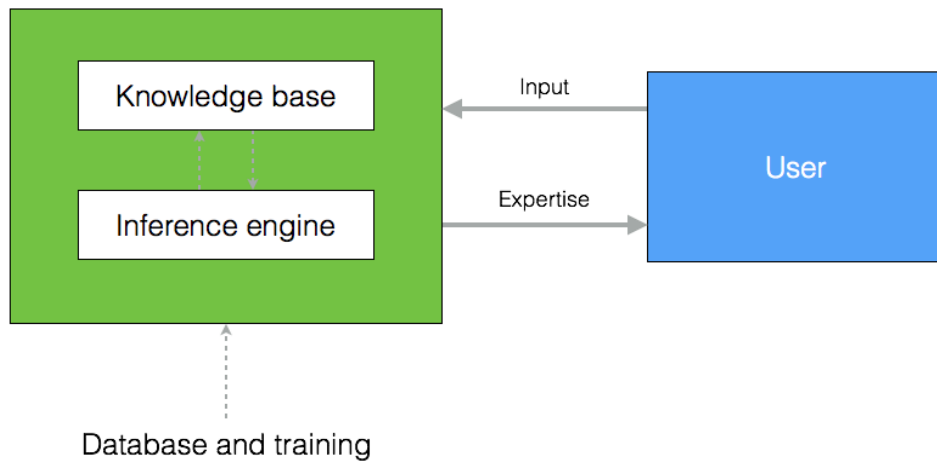
- **Artificial Intelligence (AI)** is a way to make machines think and behave **intelligently**.
- Intelligent **programs**
- We want the machines to **sense, reason, think, and act**.
- We want our machines to **be rational** too.
- AI is closely related to the **study of human brain**.
- By **mimicking** the way the **human brain** learns, thinks, and acts, we can build a machine that can do the same.

Why do we need to study AI?

- **AI can impact every aspect of our lives.**
- AI is producing **spectacular products** such as self-driving cars and intelligent robots that can walk.
- **We need AI systems that can:**
 - Handle **large amounts of data** in an efficient way.
 - Ingest data simultaneously from **multiple sources** without any lag.
 - Index and organize data in a way that allows us to **derive insights**.
 - Learn from new data and **update constantly** using the right learning algorithms.
 - Think and respond to situations based on the **conditions in real time**.

Applications of AI

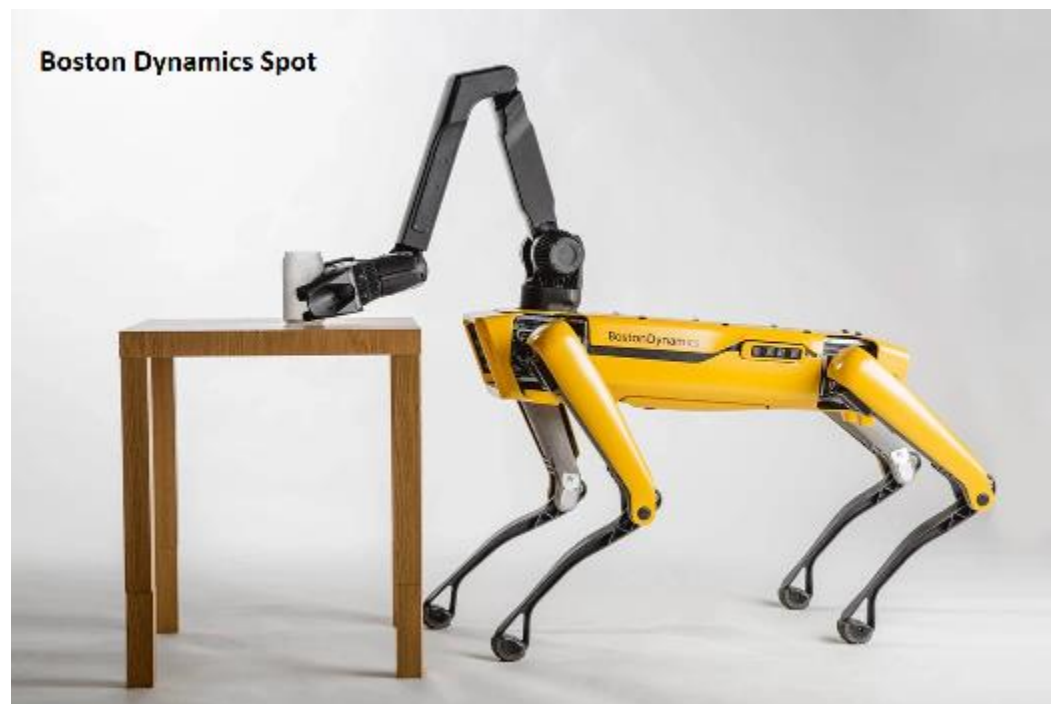
1. Computer Vision
2. Natural Language Processing
3. Speech Recognition
4. Expert Systems



Applications of AI

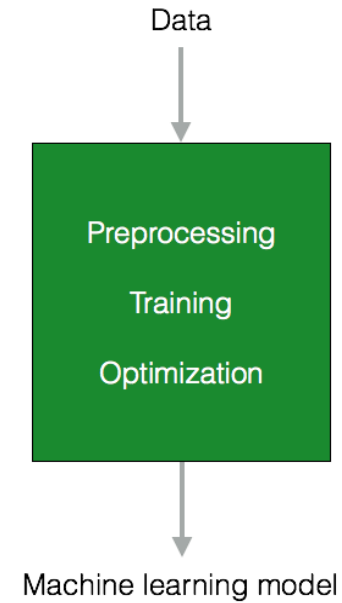
5. Games

6. Robotics

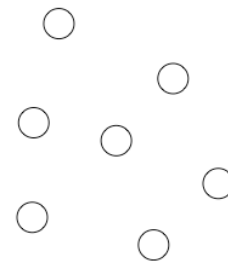


Branches of AI

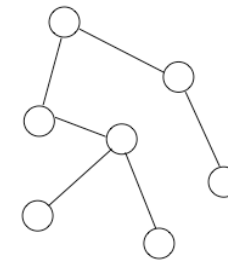
1. Machine learning and pattern recognition
2. Logic-based AI
3. Search
4. Knowledge representation
5. Planning
6. Heuristics
7. Genetic programming



Information



Knowledge

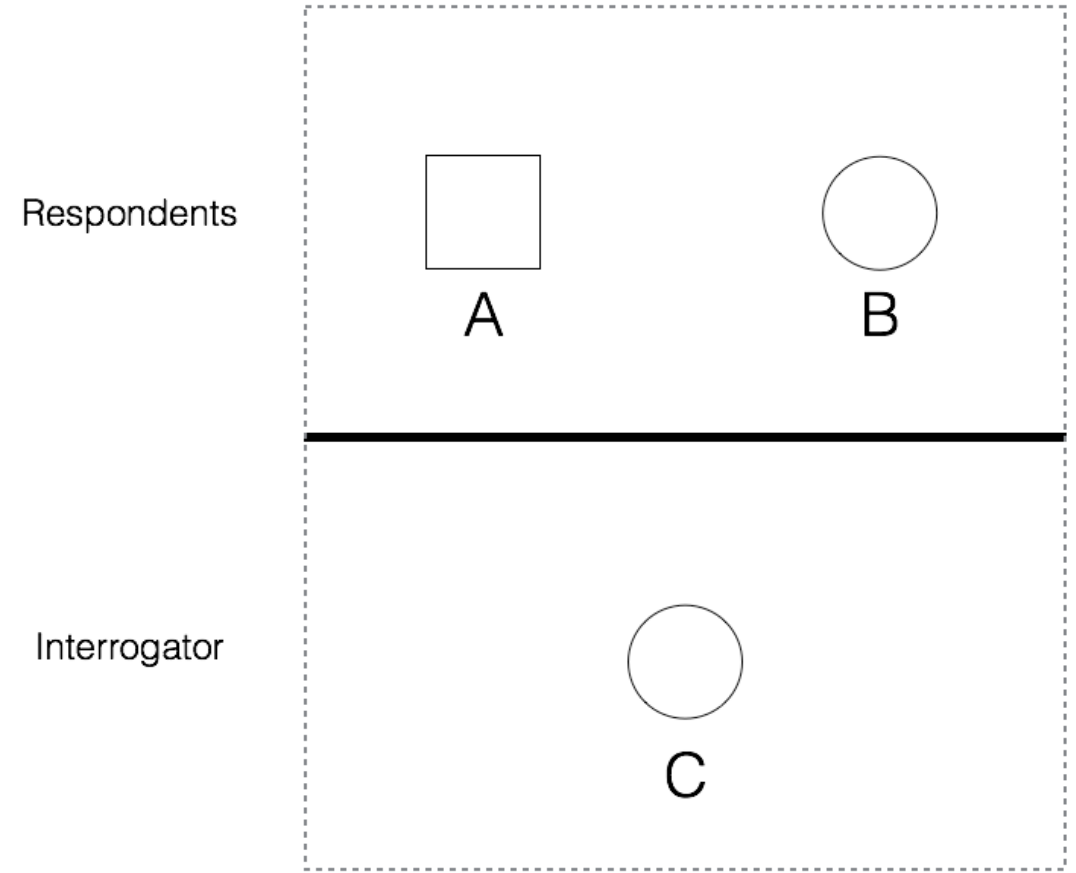


Outline

- What is AI?
- Why do we need to study AI?
- Applications of AI
- Branches of AI
- Defining intelligence using Turing Test
- Making machines think like humans
- Building rational agents
- General problem solver
- Building an intelligent agent
- Summary

Defining intelligence using Turing Test

- **Alan Turing** defined intelligent behavior as the ability to achieve human-level intelligence during a **text conversation**.
- **Difficult test, need:**
 - Natural language processing
 - Knowledge representation
 - Reasoning
 - Machine learning

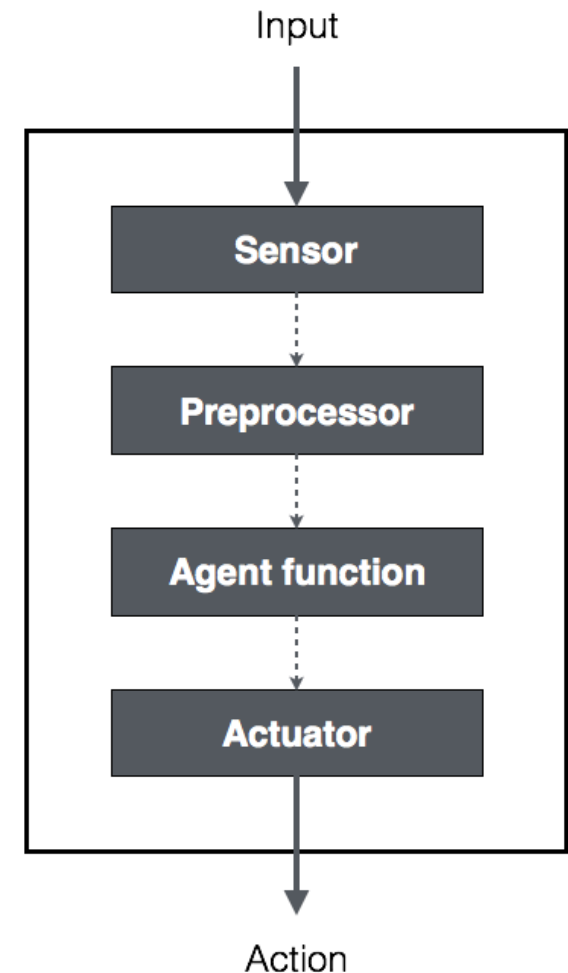


Making machines think like humans

- We need first to **understand how humans think**.
- **Cognitive Modeling** is a field of computer science that deals with simulating the **human thinking process**.
- Cognitive modeling is used in a variety of **AI applications** such as:
 - Deep learning
 - Expert systems
 - Natural language processing
 - Robotics

Building rational agents

- **Rationality** refers to doing the right thing in a given circumstance.
- An agent is said to **act rationally** if, given a set of rules, it takes actions to achieve its goals.
- **Example AI**: to design robots that can navigate unknown terrains.
- The performance depends on what **percentage of that task is complete**.



General Problem Solver

- The **General Problem Solver (GPS)** is an AI program intended to solve any general problem using the **same base algorithm**.
- Uses a language called **Information Processing Language (IPL)** to express any problem with a set of well-formed formulas.
- These formulas are part of a **directed graph** with multiple **sources** and **sinks**.
 - The **sources** refer to **axioms**
 - The **sinks** refer to the **conclusions**
- **Can solve well-defined problems**, such as proving mathematical theorems in geometry and logic.
- **Fails in the real world** because of the number of possible paths you can take.

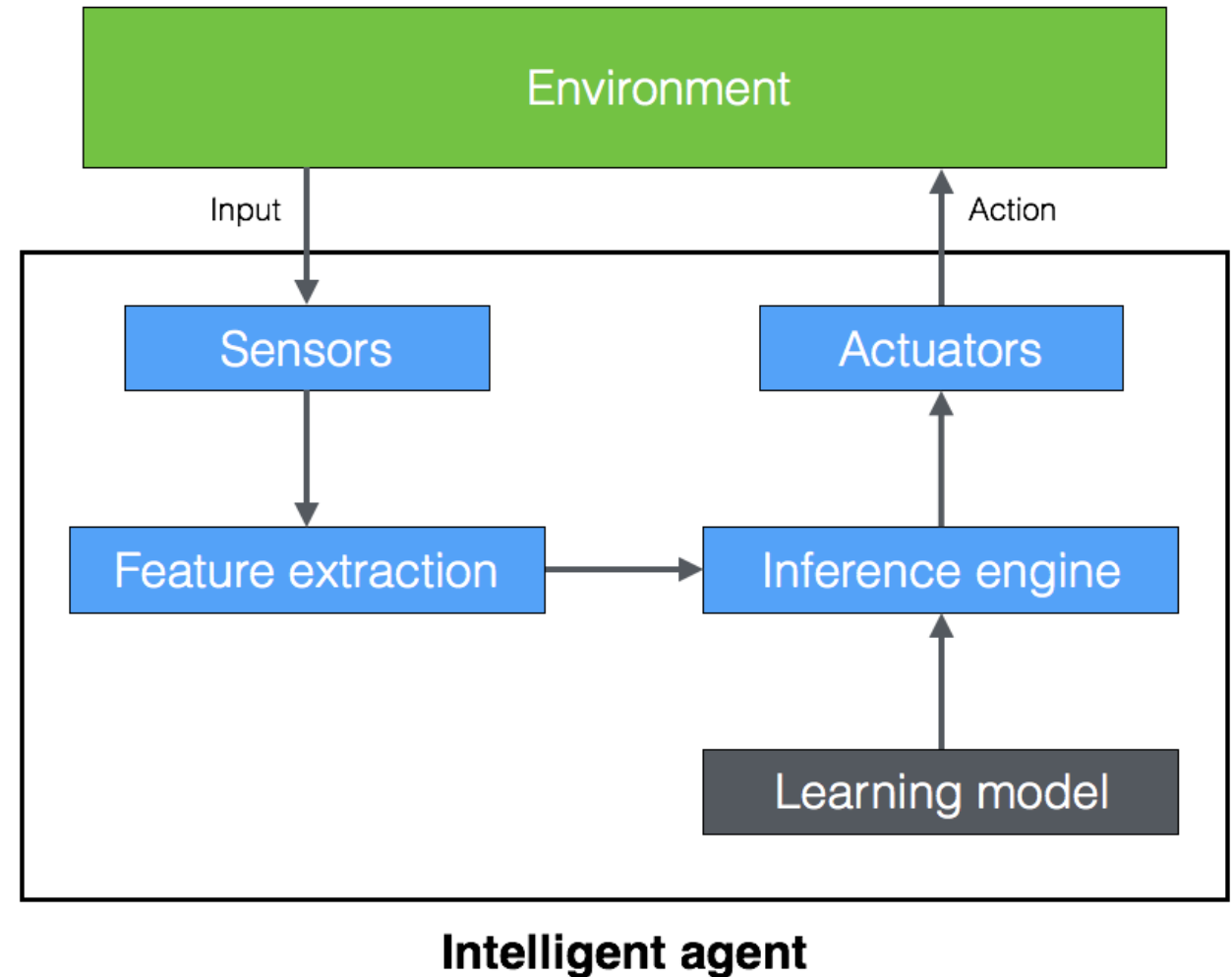
Building an intelligent agent

- **Ways** to impart intelligence to an agent:

- Machine learning
- Stored knowledge
- Rules

- **Types of Models**

- Learned models
- Analytical models



Summary

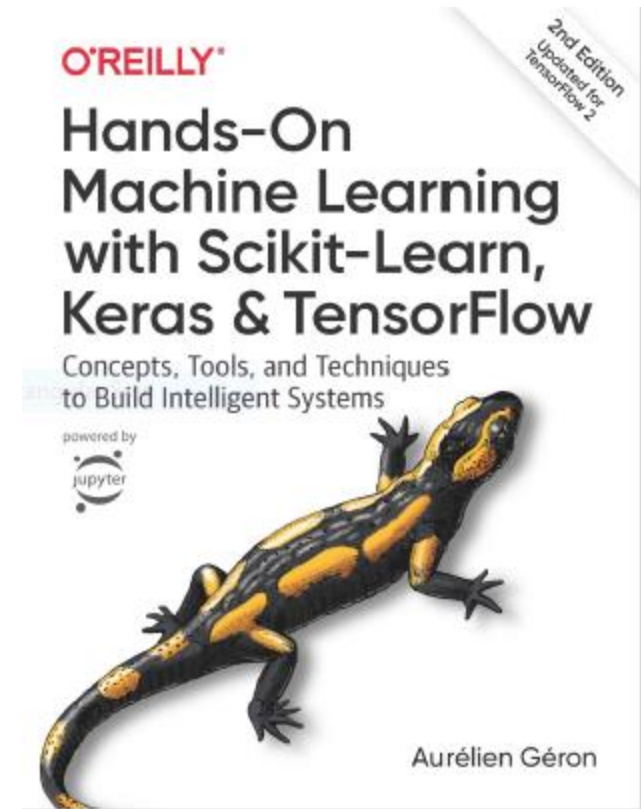
- What is AI?
- Why do we need to study AI?
- Applications of AI
- Branches of AI
- Defining intelligence using Turing Test
- Making machines think like humans
- Building rational agents
- General problem solver
- Building an intelligent agent
- Summary

Machine Learning Introduction

Prof. Gheith Abandah

Reference

- Chapter 1: **The Machine Learning Landscape**



- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>

Outline

- The Machine Learning Tsunami
- What Is Machine Learning?
- Why Use Machine Learning?
- Types of Machine Learning Systems
- Main Challenges of Machine Learning
- Testing and Validating
- Summary
- Exercises

The Machine Learning Tsunami

- YouTube Video: **From Artificial Intelligence to Superintelligence: Nick Bostrom on AI & The Future of Humanity** From Science Time

<https://youtu.be/Kktn6BPg1sl>

The Machine Learning Tsunami

- In **2006**, **Geoffrey Hinton** *et al.* published a paper showing how to **train a deep neural network** capable of recognizing handwritten digits with state-of-the-art precision (>98%). They branded this technique **Deep Learning**.
- Training a deep neural net was widely considered impossible at the time, and most researchers had abandoned the idea since the 1990s.
- Fast-forward 10 years and **ML has conquered the industry**: it is now at the heart of much of the magic in today's high-tech products.

Outline

- ✓ The Machine Learning Tsunami
 - What Is Machine Learning?
 - Why Use Machine Learning?
 - Types of Machine Learning Systems
 - Main Challenges of Machine Learning
 - Testing and Validating
 - Summary
 - Exercises

What Is Machine Learning?

- YouTube Video: **What is Machine Learning?** from Google Cloud Platform

<https://youtu.be/HcqpanDadyQ>

What Is Machine Learning?

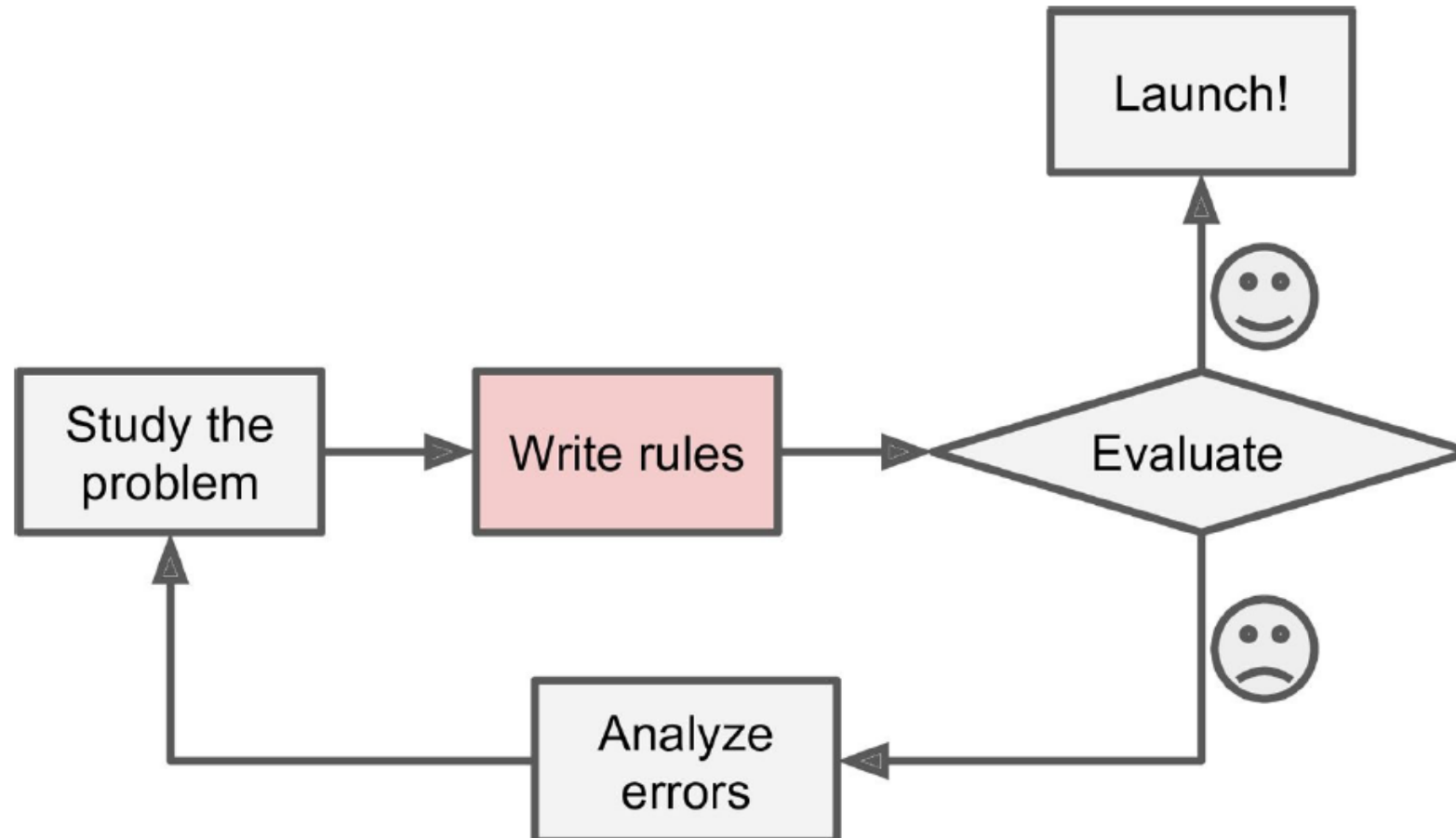
- The science (and art) of programming computers so they can **learn from data**.
- The field of study that gives computers the ability to **learn without being explicitly programmed**. Arthur Samuel, *1959*
- A computer program is said to learn from **experience E** with respect to some **task T** and some **performance** measure **P**, if its performance on T, as measured by P, **improves with experience E**. Tom Mitchell, *1997*
 - **E: Training set** made of **training instances (samples)**
 - **T: Test set**
 - **P:** Such as **accuracy**

Outline

- ✓ The Machine Learning Tsunami
- ✓ What Is Machine Learning?
 - Why Use Machine Learning?
 - Types of Machine Learning Systems
 - Main Challenges of Machine Learning
 - Testing and Validating
 - Summary
 - Exercises

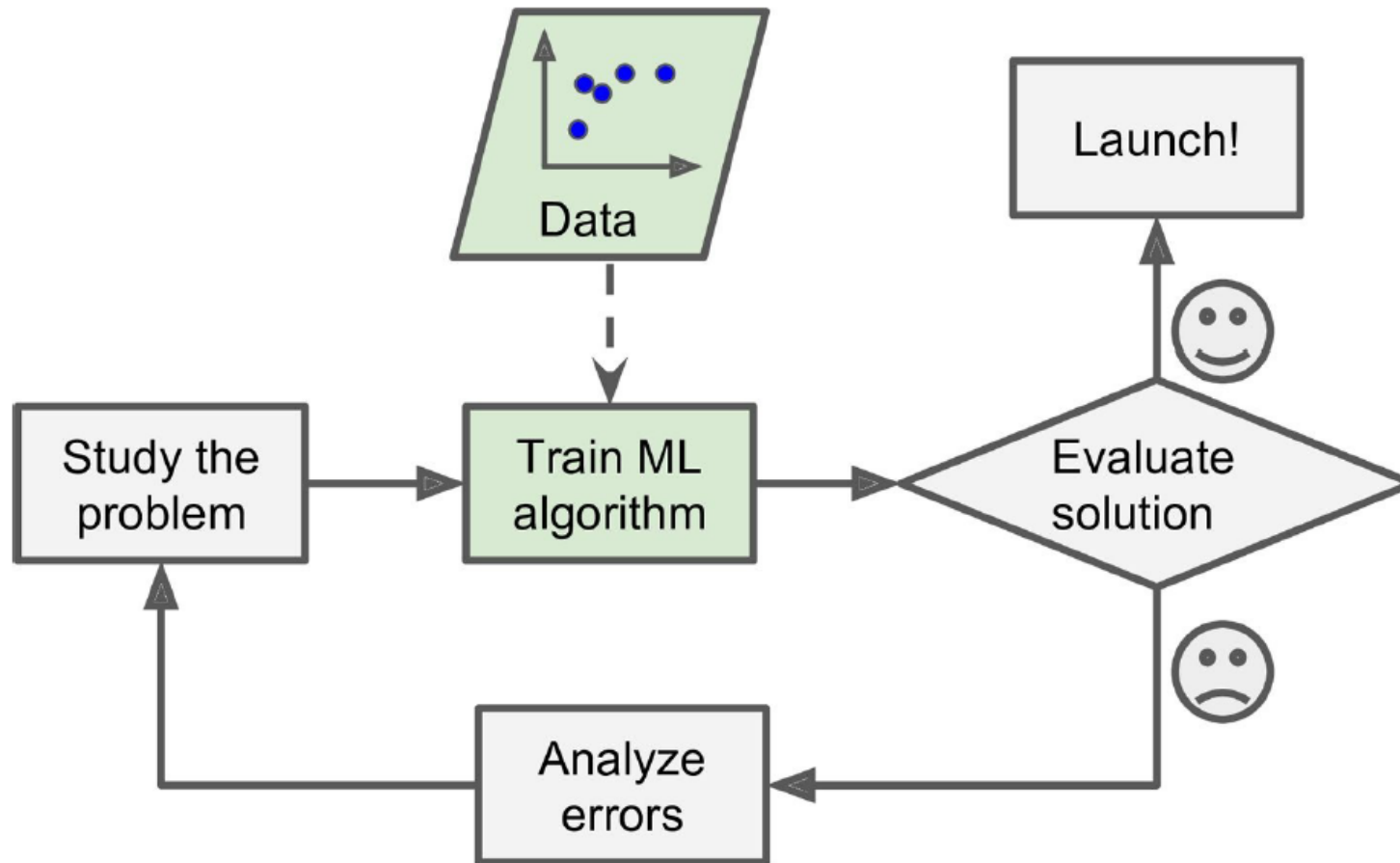
Why Use Machine Learning?

Spam filter using traditional programming techniques



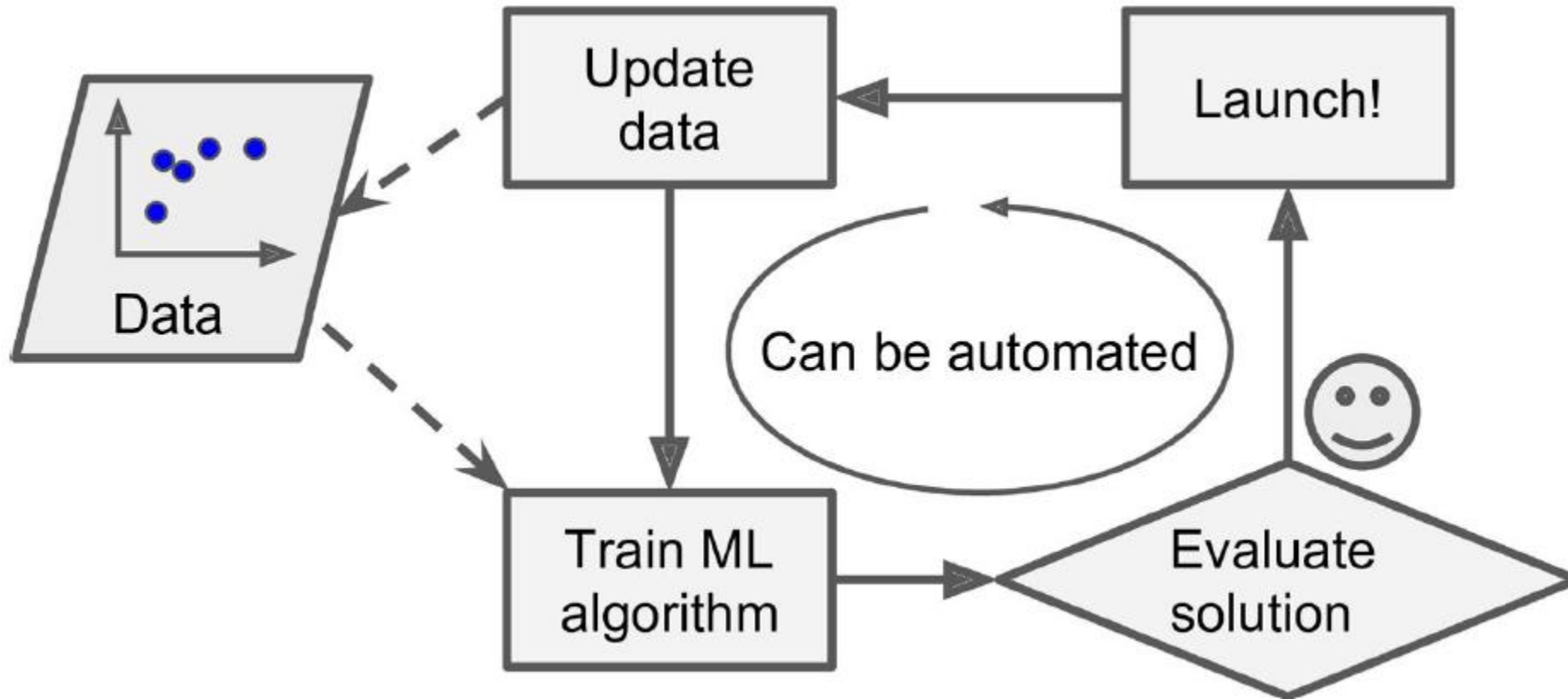
Why Use Machine Learning?

Spam filter using machine learning techniques 1/2



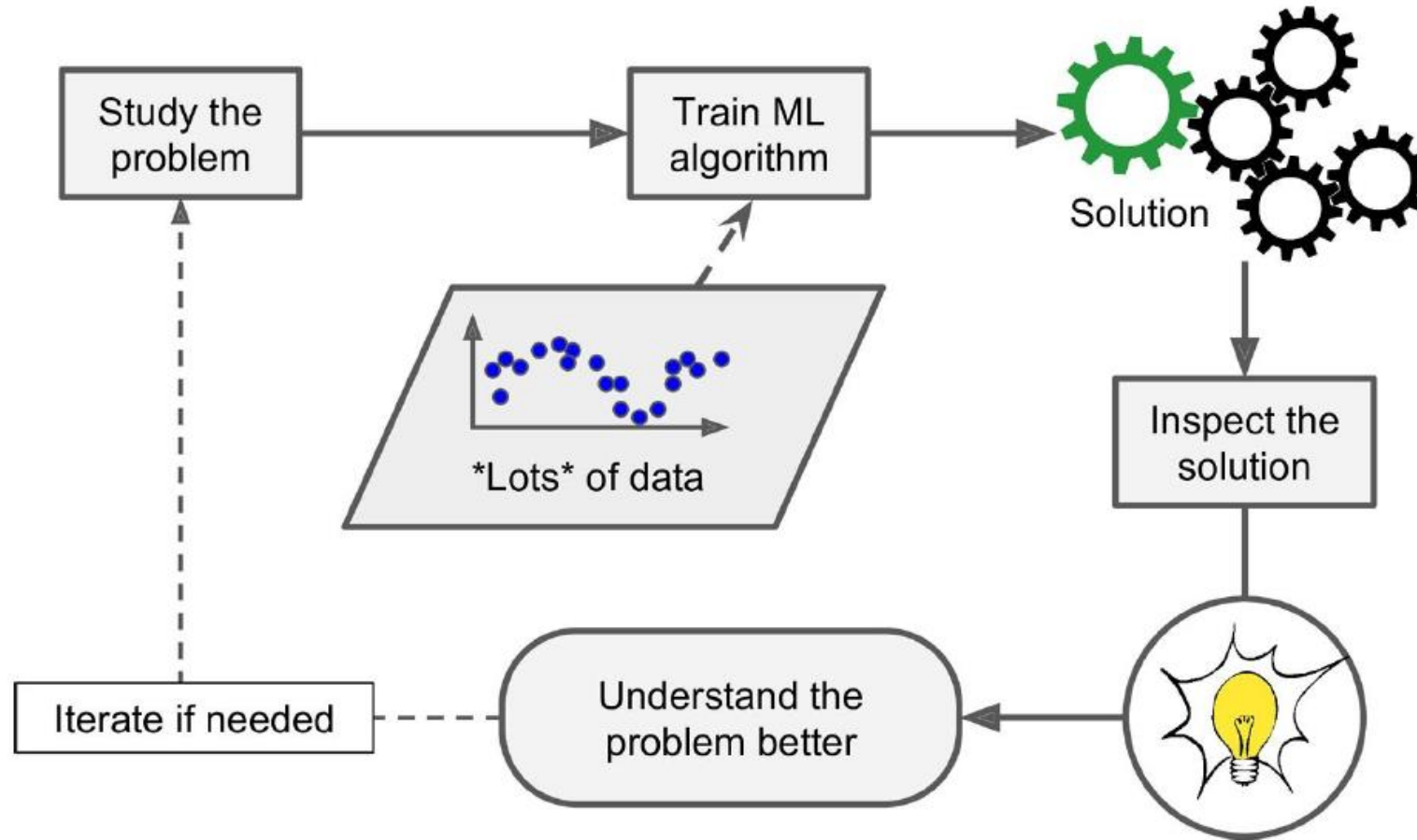
Why Use Machine Learning?

Automatically adapting to change 2/2



Why Use Machine Learning?

ML can help humans learn (Data mining)



Outline

- ✓ The Machine Learning Tsunami
- ✓ What Is Machine Learning?
- ✓ Why Use Machine Learning?
- Types of Machine Learning Systems
- Main Challenges of Machine Learning
- Testing and Validating
- Summary
- Exercises

Types of Machine Learning Systems

- **Involves human supervision?**

1. Supervised learning
2. Unsupervised learning
3. Semi-supervised learning
4. Reinforcement learning

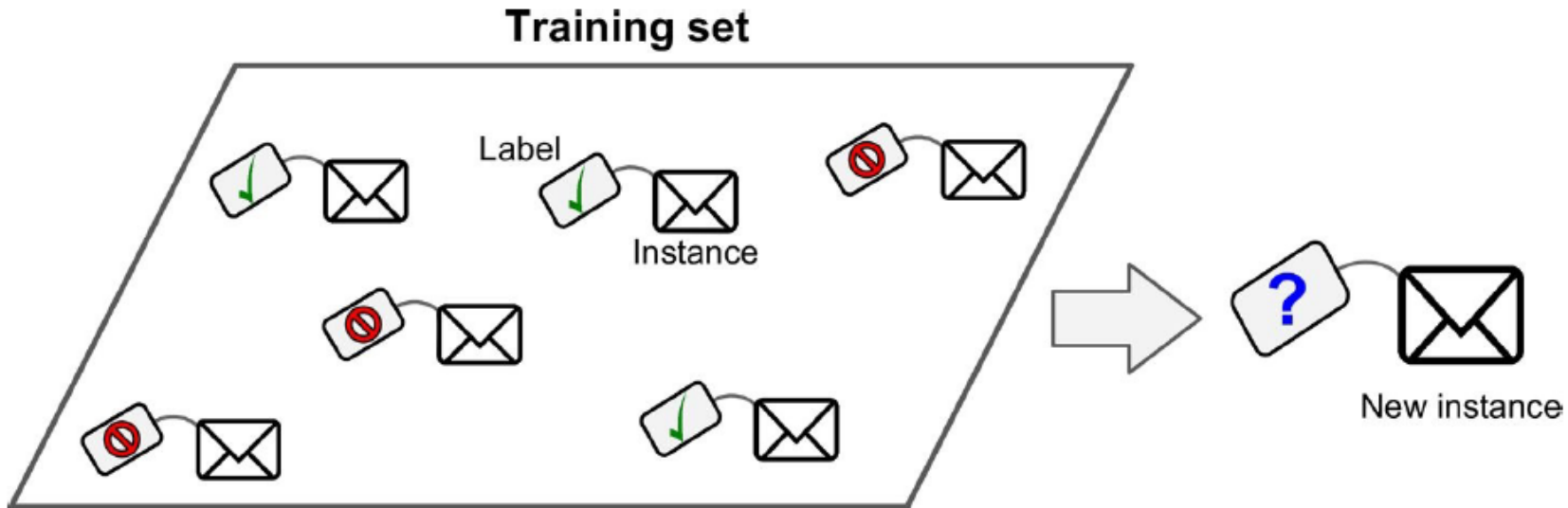
- **Learns incrementally?**

1. Batch learning
2. Online learning

- **Generalization approach**

1. Instance-based learning
2. Model-based learning

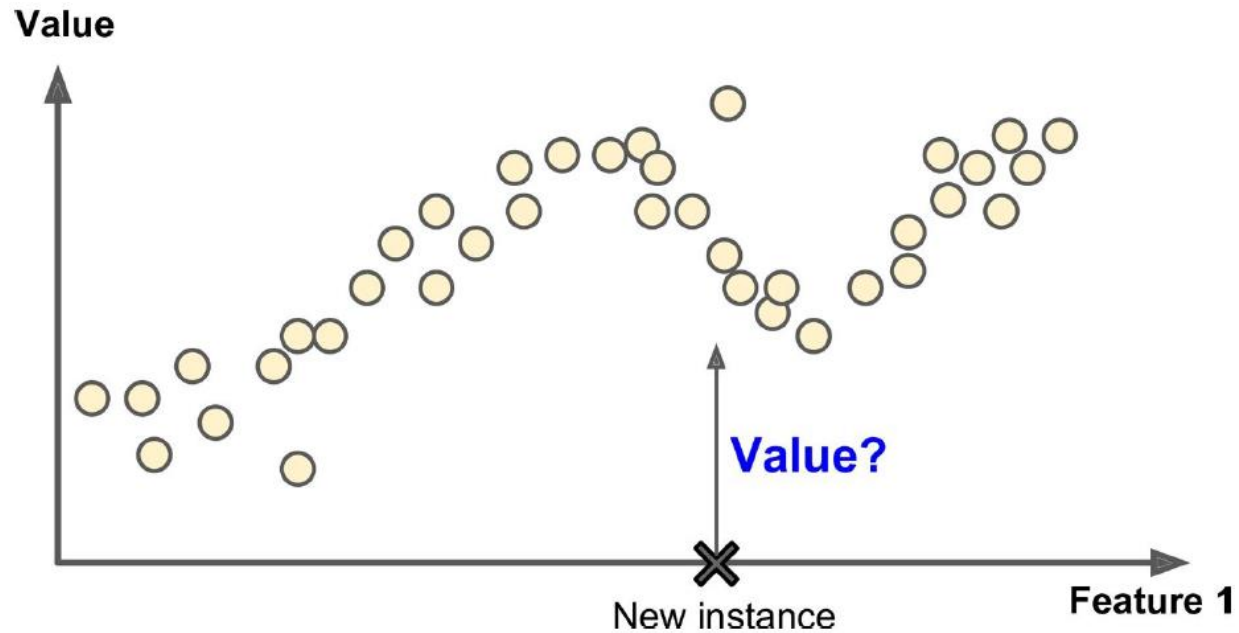
1. Supervised Learning



The training data you feed to the algorithm includes the desired solutions, called **labels**

Classification: finds the class, e.g., email type (spam or ham)

1. Supervised Learning

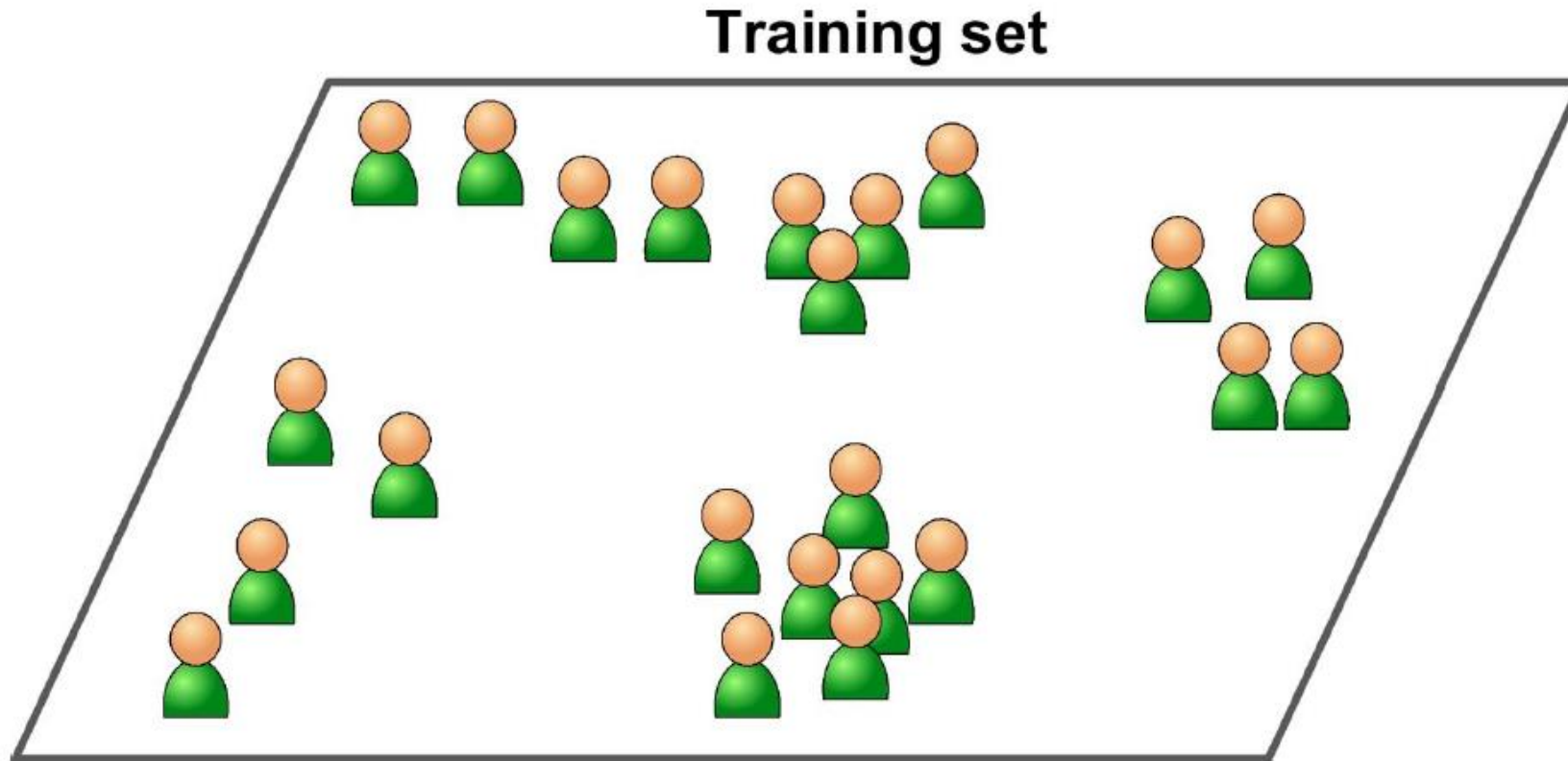


Regression: finds the value, e.g., car price

1. Supervised learning algorithms

Algorithm	Type
k-Nearest Neighbors	Both
Linear Regression	Regression
Logistic Regression	Classification
Support Vector Machines (SVMs)	Both
Decision Trees	Both
Random Forests	Both
Neural Networks	Both

2. Unsupervised Learning

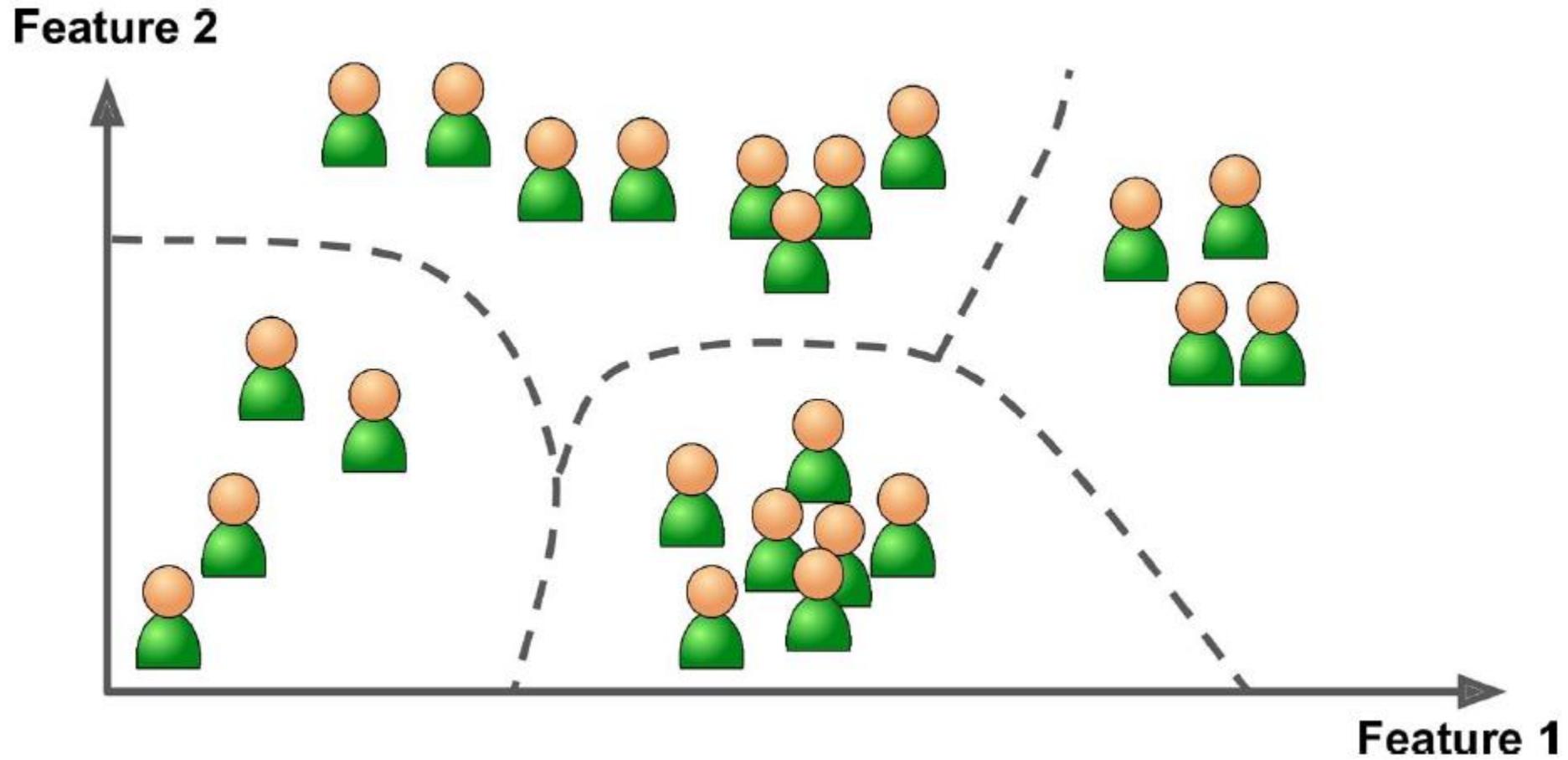


The training data is **unlabeled**.

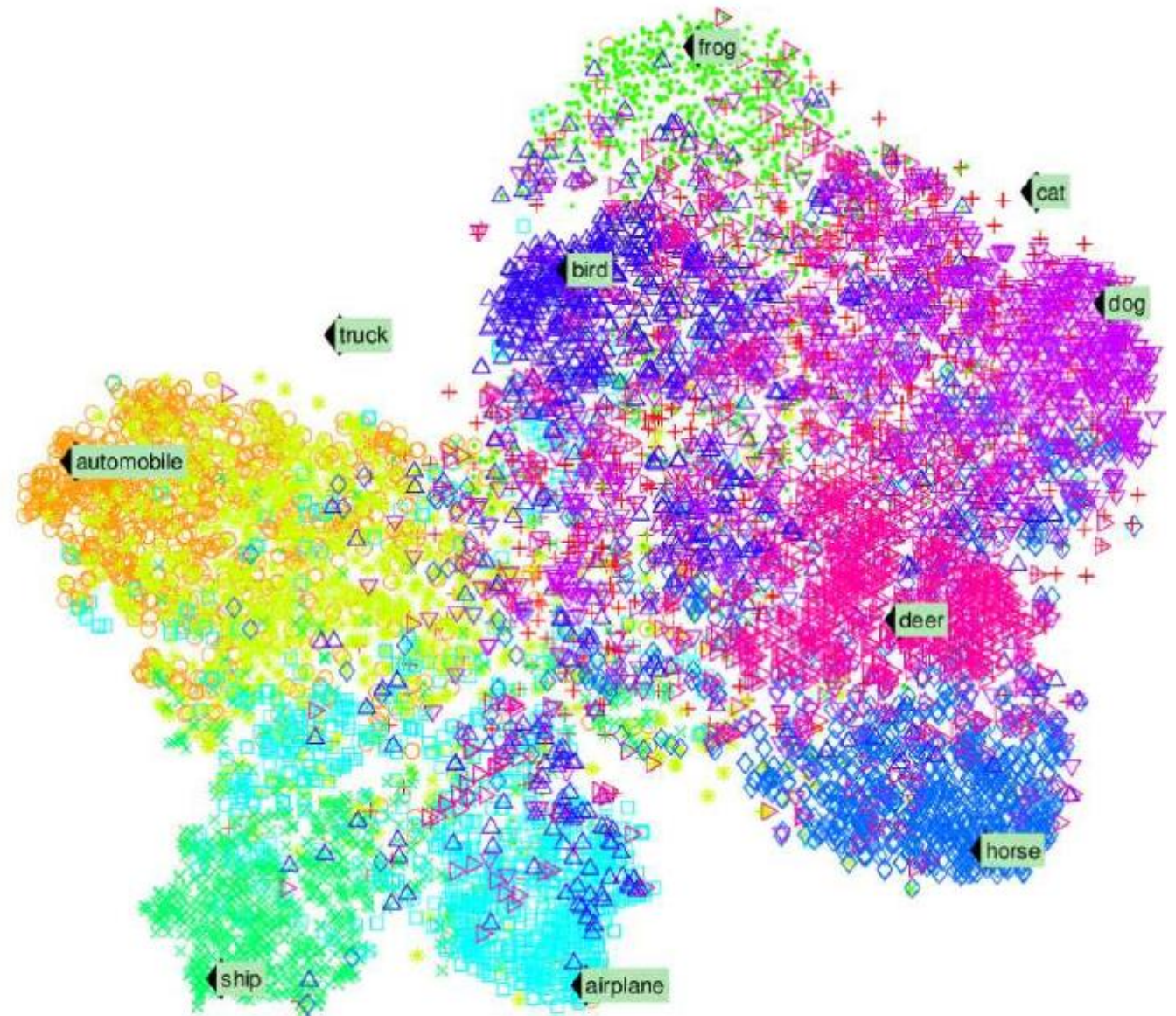
2. Unsupervised learning algorithms

- Clustering
 - k-Means
 - Hierarchical Cluster Analysis (HCA)
 - Expectation Maximization
- Visualization and dimensionality reduction
 - Principal Component Analysis (PCA)
 - Kernel PCA
 - Locally-Linear Embedding (LLE)
 - t-distributed Stochastic Neighbor Embedding (t-SNE)
- Association rule learning
 - Apriori
 - Eclat

2.a Clustering



2.b Visualization



2.c Dimensionality Reduction

- The goal is to **simplify the data** without losing too much information.
- One way to do this is to **merge** several **correlated features** into one. For example, a car's mileage may be very correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear.
- Also called **feature extraction**.

2.d Anomaly Detection



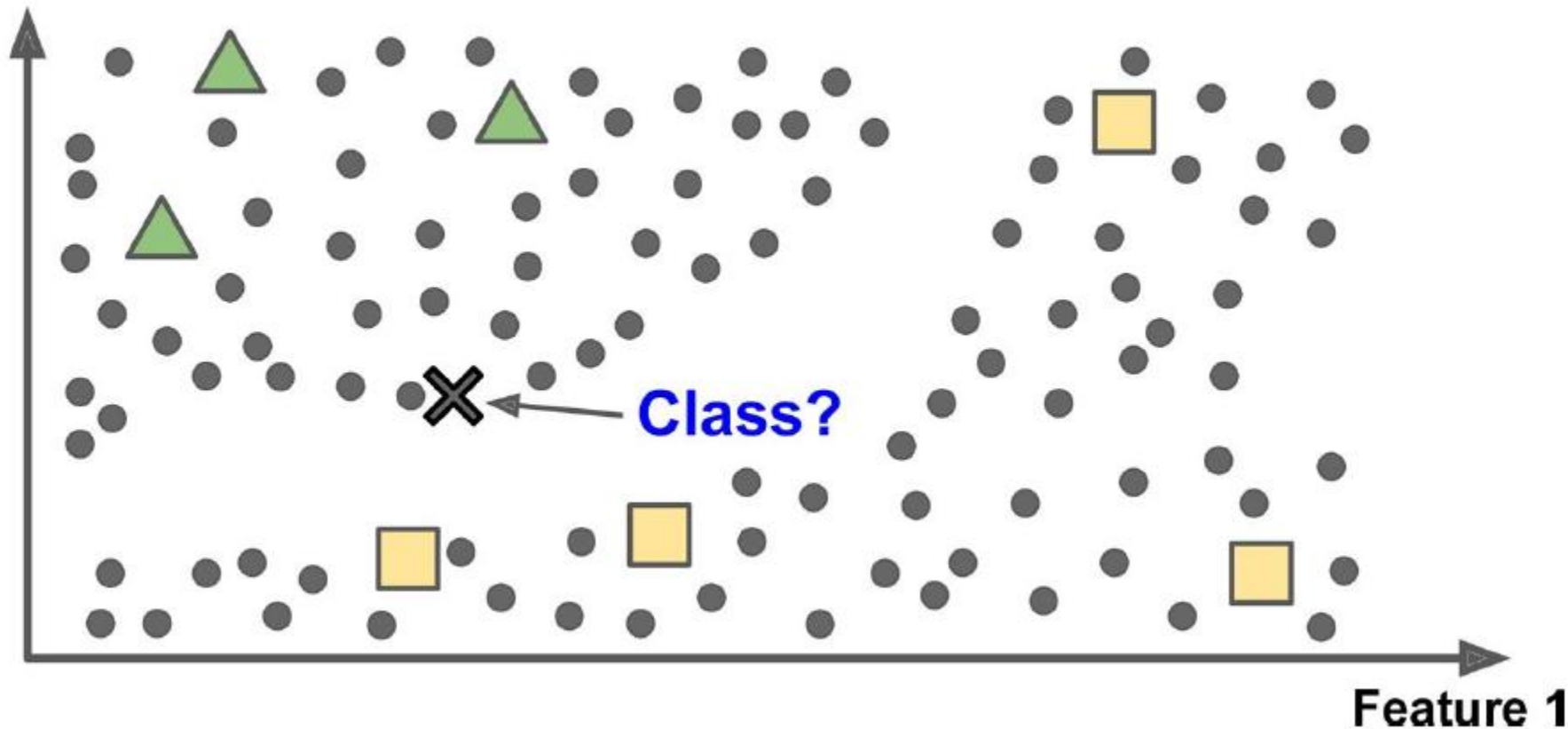
2.e Association Rule Learning

- The **goal is to dig into large amounts** of data and **discover interesting relations** between attributes.
- For example, suppose you own a supermarket. Running an association rule on your sales logs may reveal that people who purchase **barbecue sauce** and **potato chips** also tend to buy steak. Thus, you may want to place these items close to each other.

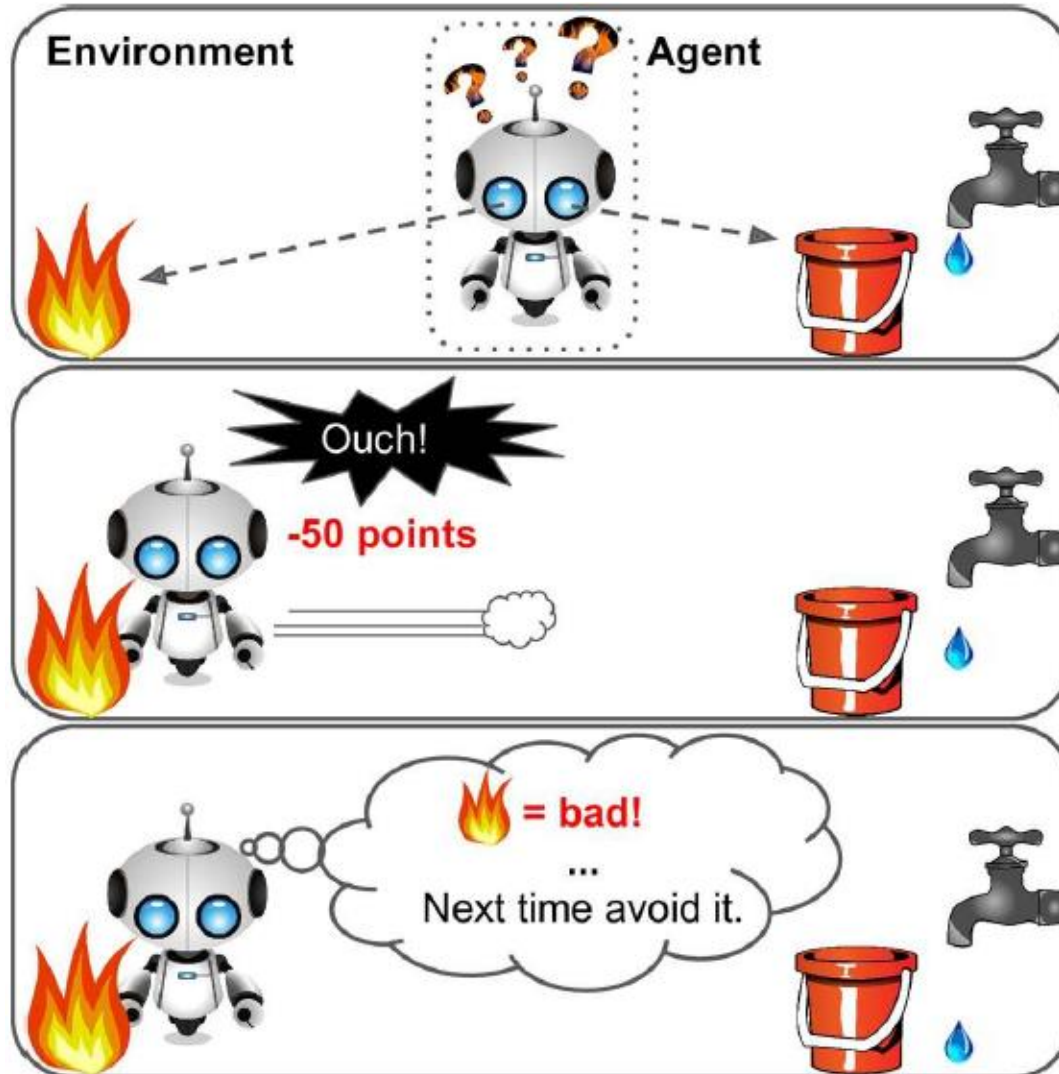
3. Semi-supervised Learning

Partially labeled training data, usually a lot of unlabeled data and a little bit of labeled data. E.g., Google Photos.

Feature 2



4. Reinforcement Learning



- 1 Observe
- 2 Select action using policy
- 3 Action!
- 4 Get reward or penalty
- 5 Update policy (learning step)
- 6 Iterate until an optimal policy is found

Types of Machine Learning Systems

✓ Involves human supervision?

1. Supervised learning
2. Unsupervised learning
3. Semi-supervised learning
4. Reinforcement learning

• Learns incrementally?

1. Batch learning
2. Online learning

• Generalization approach

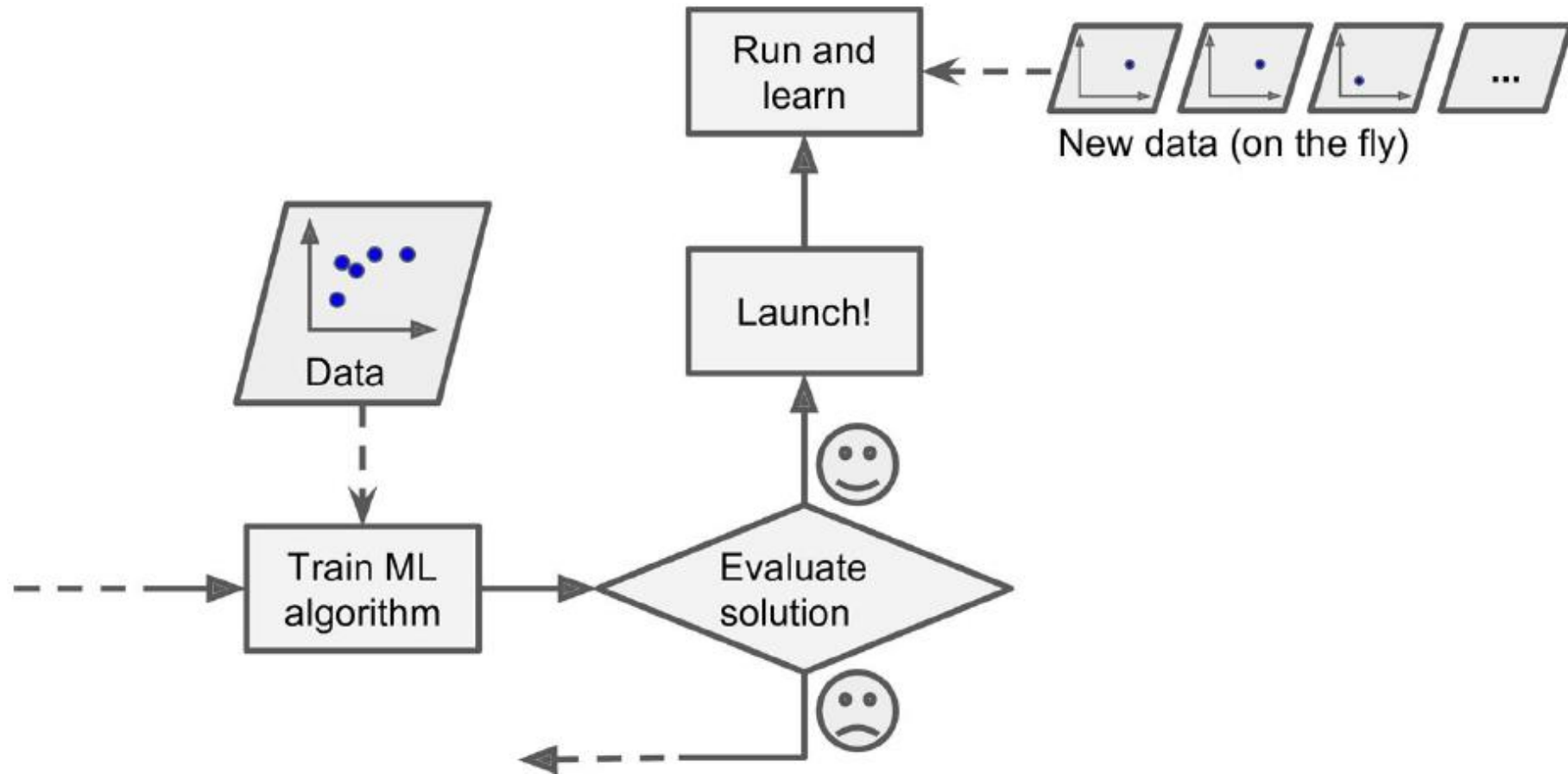
1. Instance-based learning
2. Model-based learning

1. Batch (offline) Learning

- Must be **trained** using **all the available data**.
- This will generally take a **lot of time** and computing resources, so it is typically done **offline**.
- First the system is **trained**, and **then** it is **launched** into production and runs without learning anymore; it just applies what it has learned.

2. Online Learning

Examples: Stock prices, huge data



Types of Machine Learning Systems

✓ Involves human supervision?

1. Supervised learning
2. Unsupervised learning
3. Semi-supervised learning
4. Reinforcement learning

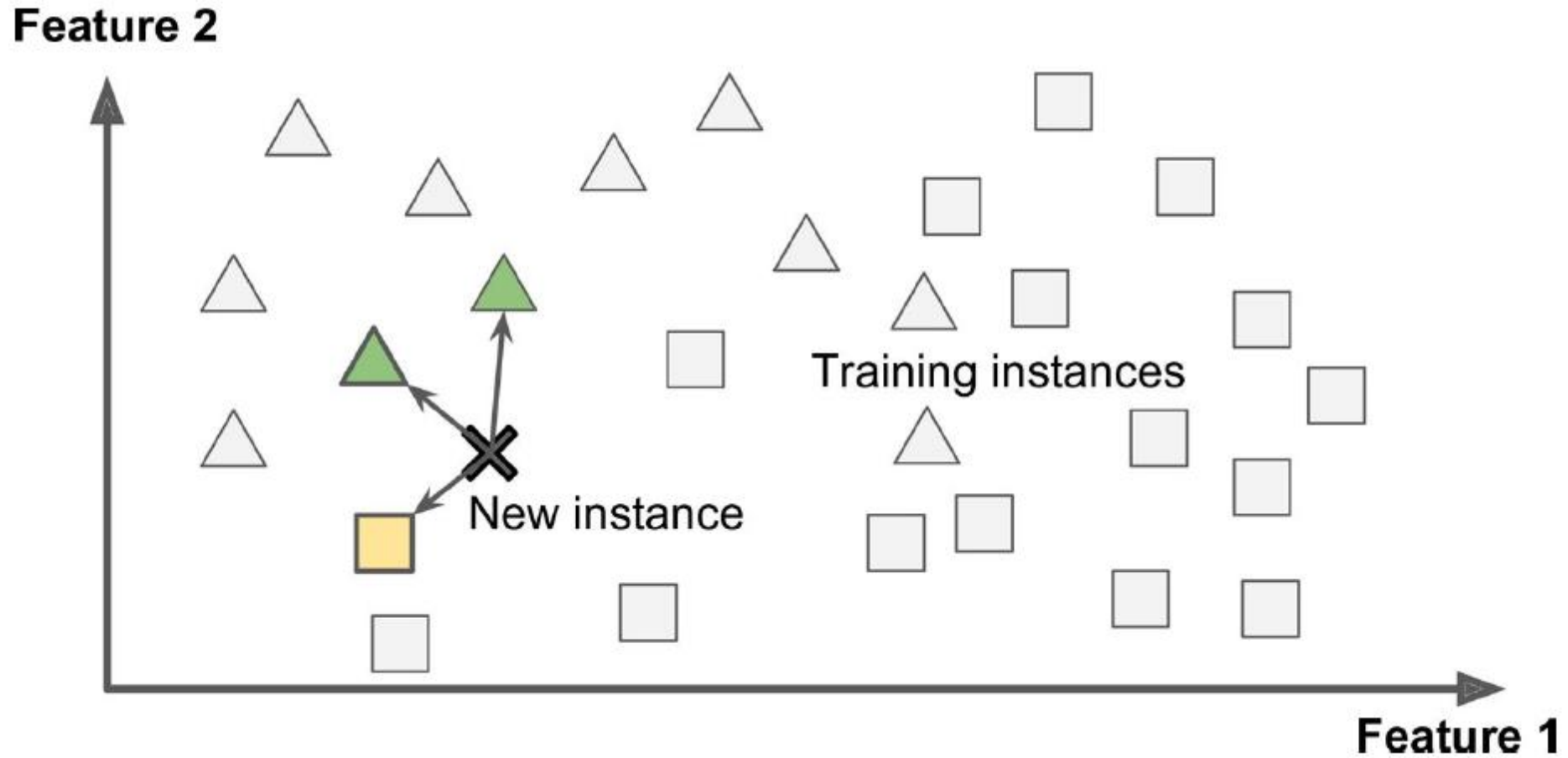
✓ Learns incrementally?

1. Batch learning
2. Online learning

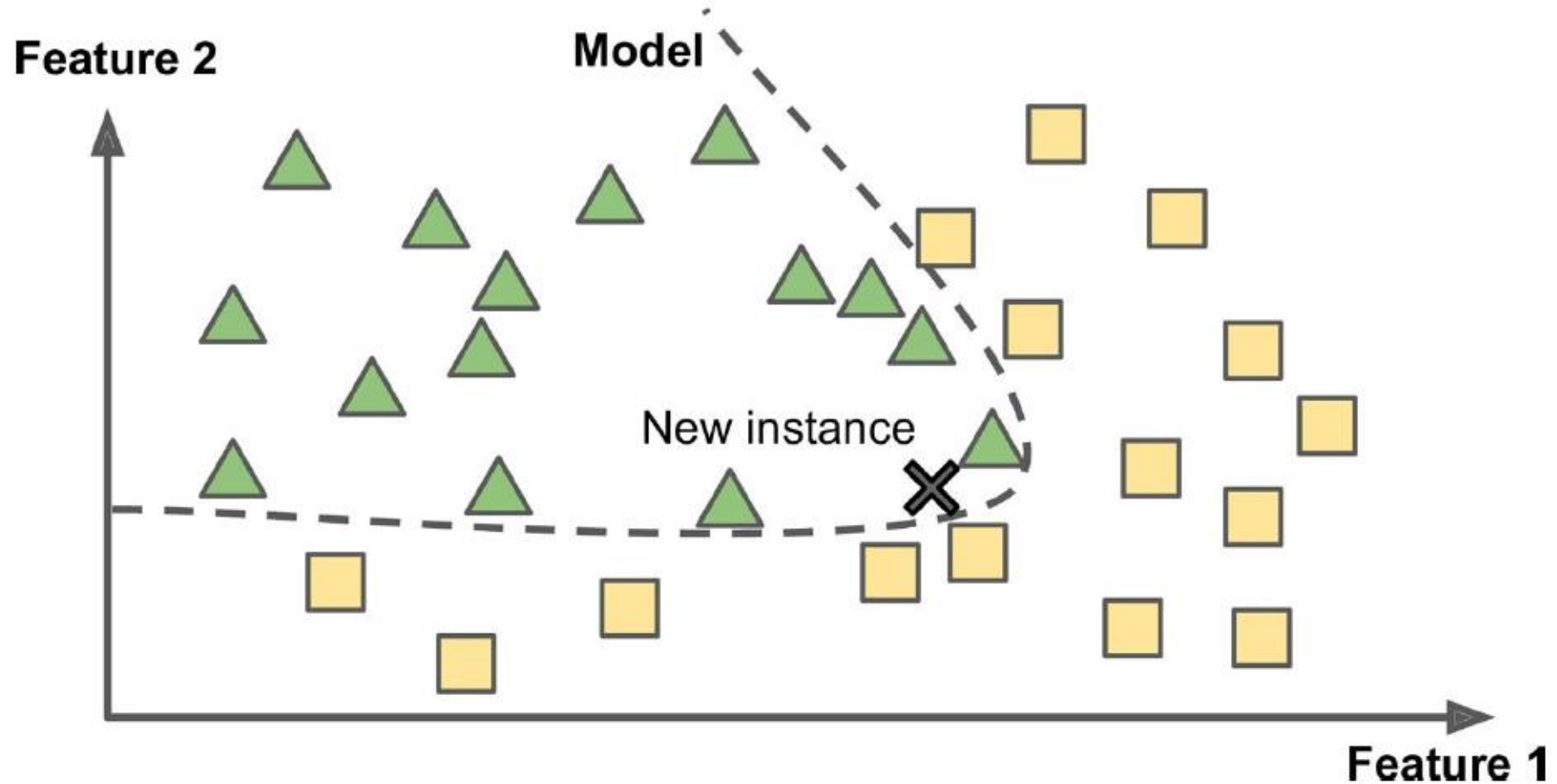
• Generalization approach

1. Instance-based learning
2. Model-based learning

1. Instance-based Learning



2. Model-based Learning

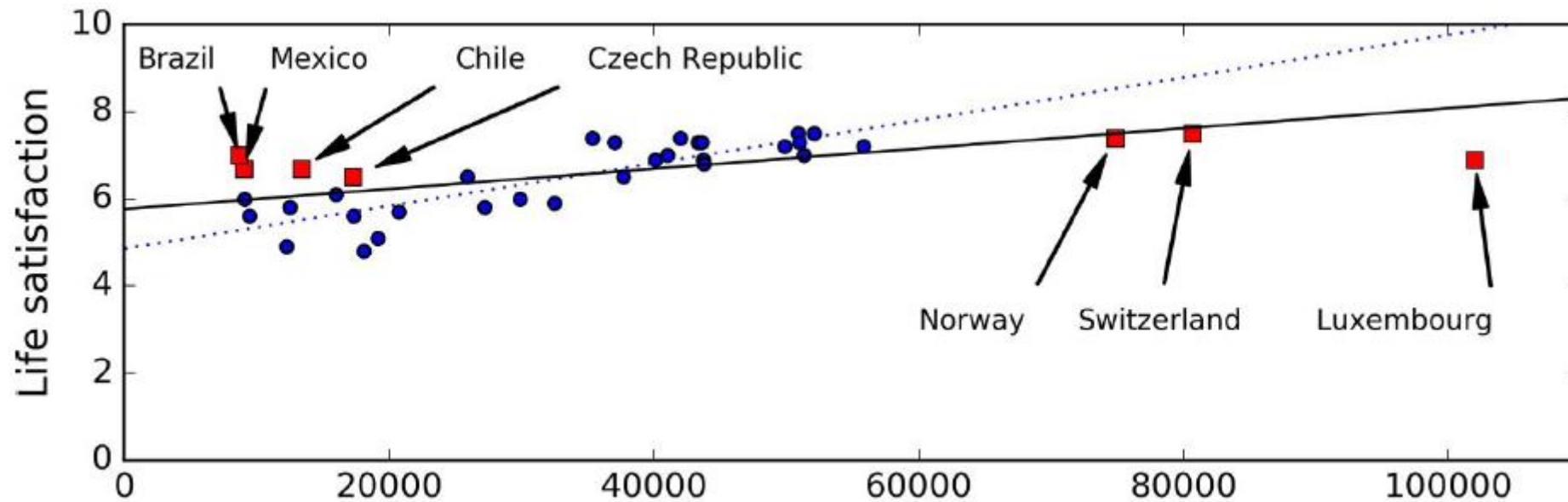


Outline

- ✓ The Machine Learning Tsunami
- ✓ What Is Machine Learning?
- ✓ Why Use Machine Learning?
- ✓ Types of Machine Learning Systems
 - Main Challenges of Machine Learning
 - Testing and Validating
 - Summary
 - Exercises

Main Challenges of Machine Learning (due to bad data)

1. **Insufficient** quantity of training data
2. **Non-representative** training data



Main Challenges of Machine Learning (due to bad data)

3. **Poor-quality** data that contains:

- Errors
- Outliers
- Noise

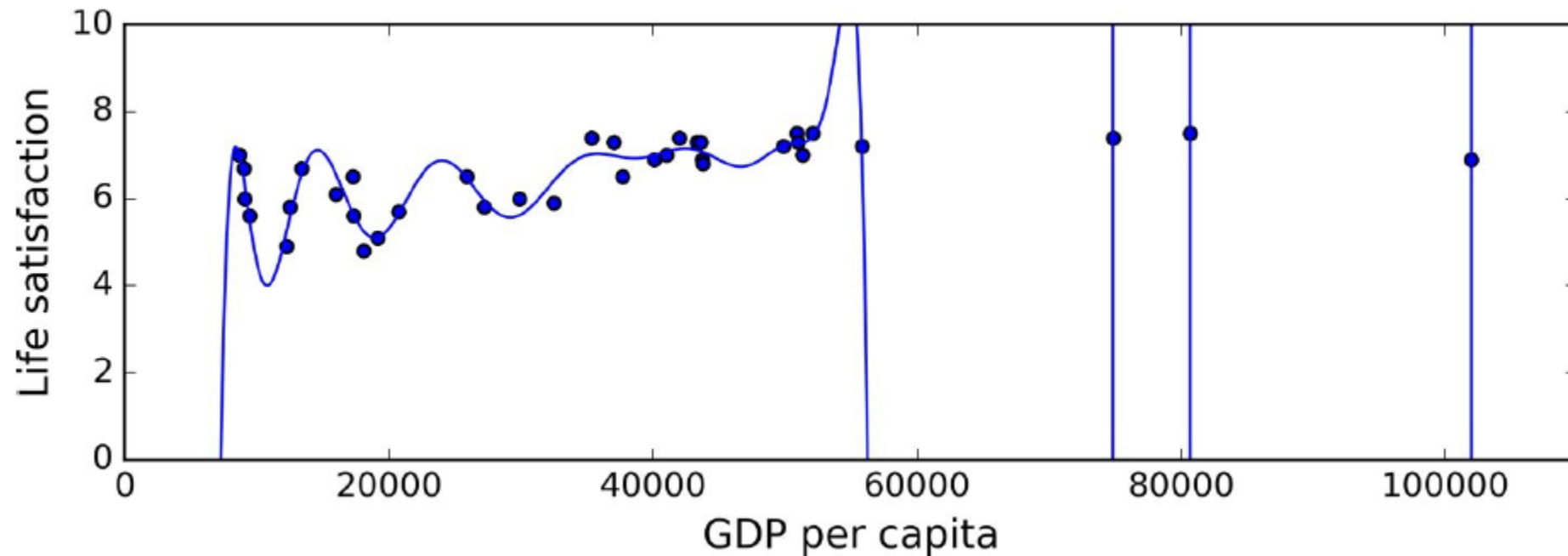
4. **Irrelevant features**: Need **feature engineering**:

- **Feature selection**: selecting the most useful features.
- **Feature extraction**: combining existing features to produce a more useful one.
- **Creating new features** by gathering new data.

Main Challenges of Machine Learning (due to bad algorithm)

1. **Overfitting** the training data

- **Regularization** constrains the model's **hyperparameters** to make it simpler and reduce the risk of overfitting.



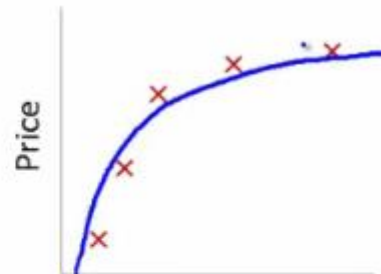
Main Challenges of Machine Learning (due to bad algorithm)

2. Under-fitting the training data



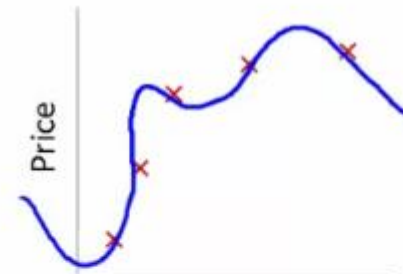
$$\theta_0 + \theta_1 x$$

High bias
(underfit)



$$\theta_0 + \theta_1 x + \theta_2 x^2$$

"Just right"



$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

High variance
(overfit)

Outline

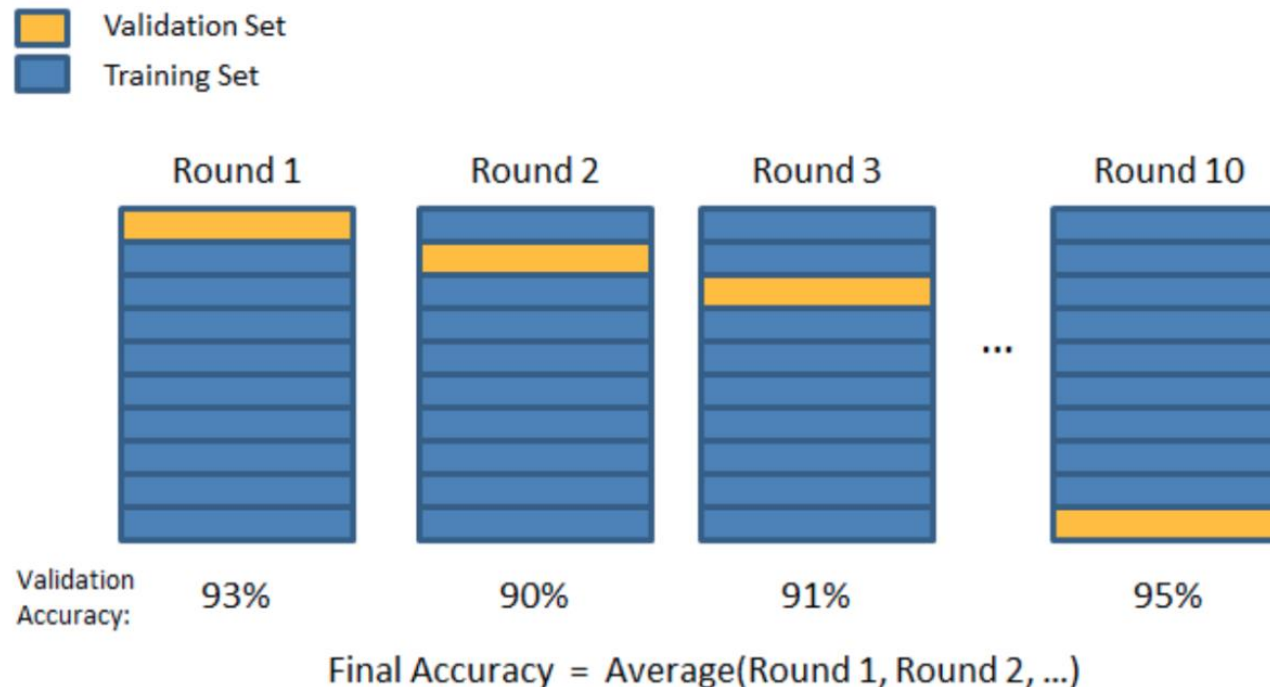
- ✓ The Machine Learning Tsunami
- ✓ What Is Machine Learning?
- ✓ Why Use Machine Learning?
- ✓ Types of Machine Learning Systems
- ✓ Main Challenges of Machine Learning
 - Testing and Validating
 - Summary
 - Exercises

Testing and Validating

- **Split** your data into two sets (**cross validation**):
 - The **training set** (80%)
 - The **test set** (20%)
- **Evaluate**:
 - The training error
 - The generalization error
- If the training error is low but the generalization error is high, it means that your model is overfitting the training data.
- When the ML algorithm is iterative, often we use a third set: **validation set**.

Cross Validation

- In **k-fold cross-validation**, the original sample is randomly partitioned into **k** equal size subsamples.



Summary

- ML is about making machines get better at some task by learning from data, instead of having to explicitly code rules.
- Types of ML systems: supervised or not, batch or online, and instance-based or model-based.
- A model-based algorithm tunes some parameters to fit the model to the training set, and then hopefully it will be able to make good predictions on new cases.
- An instance-based algorithm learns the examples by heart and uses a similarity measure to generalize to new instances.
- The system will not perform well if your training set is too small, not representative, noisy, or polluted with irrelevant features.
- Your model needs to be neither too simple (under-fit) nor too complex (over-fit).

Exercises

- How would you define Machine Learning?
- What is a labeled training set?
- Can you name four common unsupervised tasks?
- What type of Machine Learning algorithm would you use to allow a robot to walk in various unknown terrains?
- What type of algorithm would you use to segment your customers into multiple groups?
- What is an online learning system?
- What is the difference between a model parameter and a learning algorithm's hyperparameter?
- If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?
- What is the purpose of a validation set?

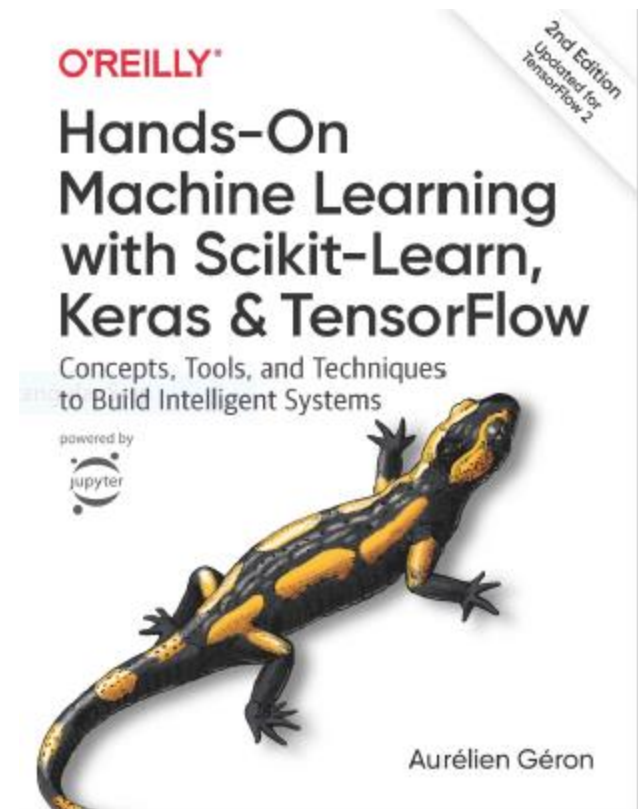
End-to-End Machine Learning Project

Prof. Gheith Abandah

Reference

- Chapter 2: **End-to-End Machine Learning Project**

- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>



The 7 Steps of Machine Learning

- YouTube Video: **The 7 Steps of Machine Learning** from Google Cloud Platform

<https://youtu.be/nKW8Ndu7Mjw>

Caution: *Alcohol is forbidden in the Islamic religion and causes addiction and has negative effects on health.*

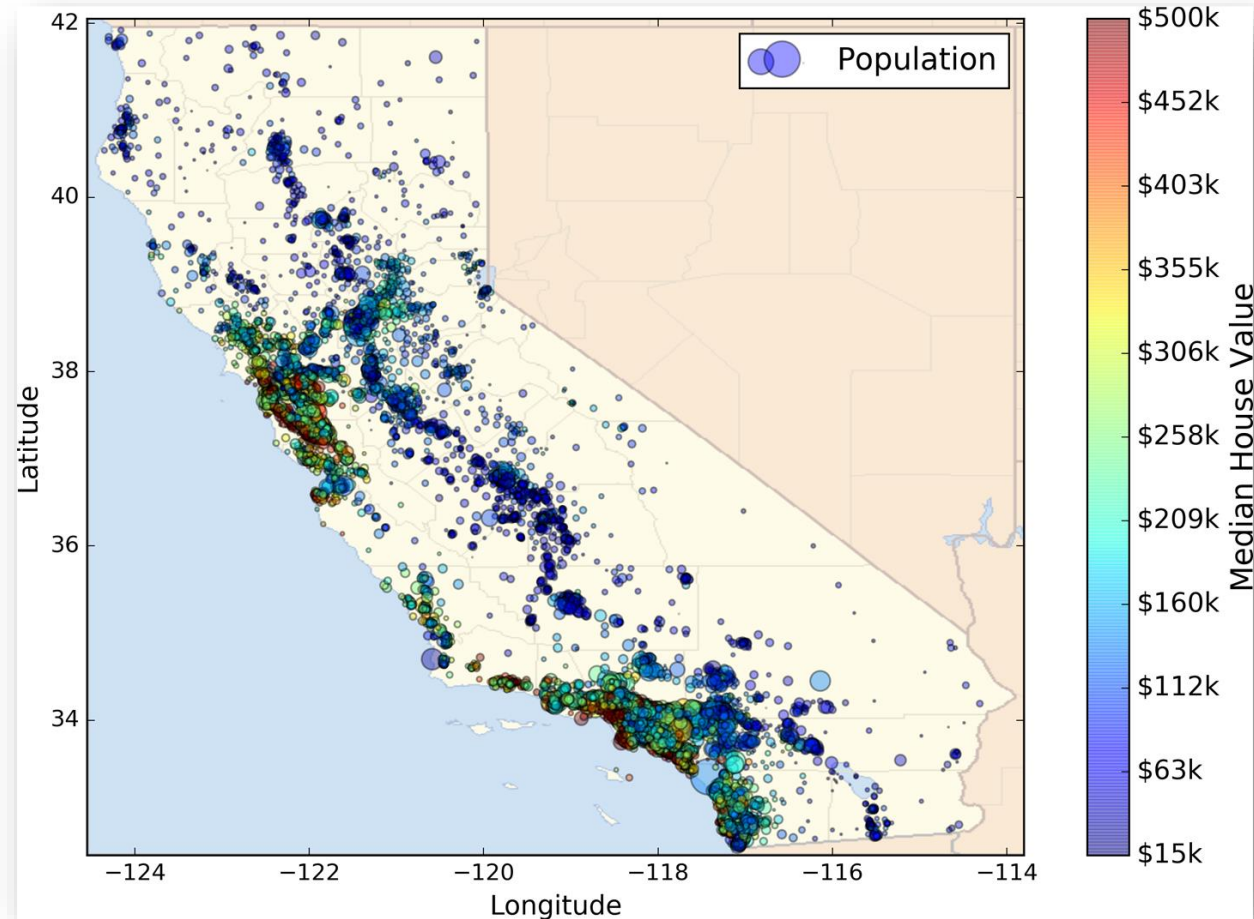
Outline

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

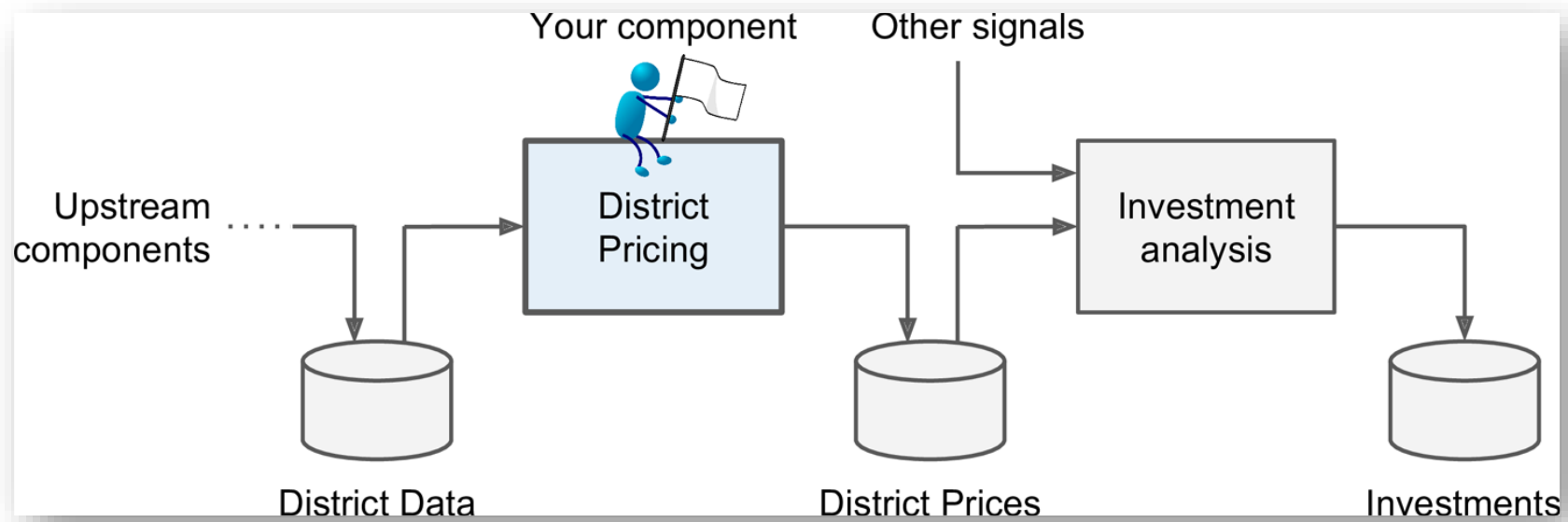
Working with Real Data

- Popular open data repositories:
 - [Tensorflow Datasets \(GitHub\)](#)
 - [UC Irvine Machine Learning Repository](#)
 - [Kaggle datasets](#)
 - [Amazon's AWS datasets](#)
 - [IEEE DataPort](#)
- Meta portals (they list open data repositories):
 - [Google Dataset Search](#)
 - <http://dataportals.org/>
 - <http://opendatamonitor.eu/>
 - <http://quandl.com/>
- Other pages listing many popular open data repositories:
 - [Wikipedia's list of Machine Learning datasets](#)
 - [Quora.com question](#)
 - [Datasets subreddit](#)

1. Look at the Big Picture: CA Housing Data



1.1. Frame the Problem

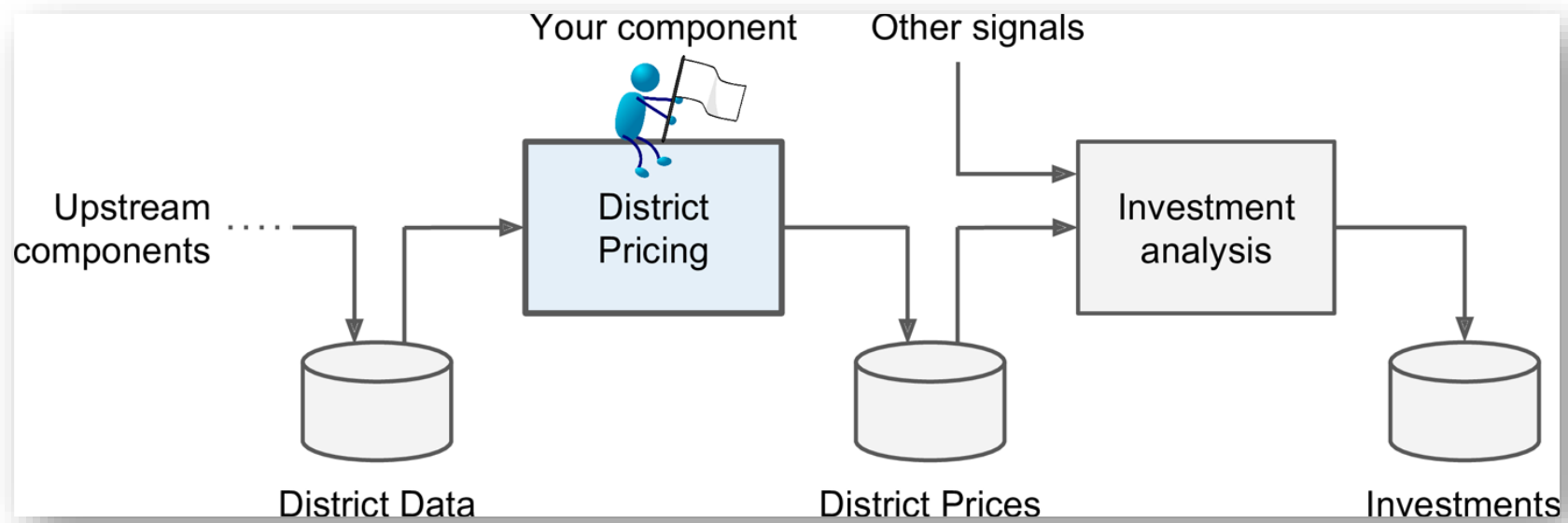


Is it supervised, unsupervised, or Reinforcement Learning?

Is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques?

Instance-based or Model-based learning?

1.1. Frame the Problem



Is it **supervised**, unsupervised, or Reinforcement Learning?

Is it a classification task, a **regression** task, or something else? Should you use **batch** learning or online learning techniques?

Instance-based or **Model-based** learning?

1.2. Select a Performance Measure

- **Root Mean Square Error (RMSE)**

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m \left(h(\mathbf{x}^{(i)}) - y^{(i)} \right)^2}$$

- m is the number of samples
- $\mathbf{x}^{(i)}$ is the feature vector of Sample i
- $y^{(i)}$ is the label or desired output
- \mathbf{X} is a matrix containing all the feature values

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

1.2. Select a Performance Measure

- **Mean Absolute Error**

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m \left| h(\mathbf{x}^{(i)}) - y^{(i)} \right|$$

- MAE is better than RMSE when there are outlier samples.

Outline

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

2. Get the Data

- If you didn't do it before, it is time now to **download** the **Jupyter notebooks** of the textbook from

<https://github.com/ageron/handson-ml2>

- Start Jupyter notebook and open [Chapter 2 notebook](#).
- Hint: If you get kernel connection problem, try
`C:\>jupyter notebook -port 8889`
- The following slides summarize the code used in this notebook.

2. Get the Data

1. Download the `housing.tgz` file from **Github** using `urllib.request.urlretrieve()` from the `urllib` package
2. Extract the data from this compressed tar file using `tarfile.open()` and `extractall()`. The data will be in the CSV file `housing.csv`
3. Read the CSV file into a Pandas DataFrame called `housing` using `pandas.read_csv()`

2.1. Take a Quick Look at the Data Structure

- Display the top five rows using the DataFrame's `head()` method
- The `info()` method is useful to get a quick description of the data
- To find categories and repetitions of some column use `housing['key'].value_counts()`
- The `describe()` method shows a summary of the numerical attributes.
- Show histogram using the `hist()` method and `matplotlib.pyplot.show()`

```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 20640 entries, 0 to 20639  
Data columns (total 10 columns):  
longitude          20640 non-null float64  
latitude           20640 non-null float64  
housing_median_age 20640 non-null float64  
total_rooms        20640 non-null float64  
total_bedrooms     20433 non-null float64  
population         20640 non-null float64  
households         20640 non-null float64  
median_income      20640 non-null float64  
median_house_value 20640 non-null float64  
ocean_proximity    20640 non-null object  
dtypes: float64(9), object(1)  
memory usage: 1.6+ MB
```

207 missing
features

```
>>> housing["ocean_proximity"].value_counts()  
<1H OCEAN      9136  
INLAND         6551  
NEAR OCEAN     2658  
NEAR BAY       2290  
ISLAND          5  
Name: ocean_proximity, dtype: int64
```

2.2. Create a Test Set

- **Split** the available data randomly to:
 - Training set (80%)
 - Test set (20%)
- The example defines a function called `split_train_test()` for illustration.
- Scikit-Learn has `train_test_split()`.
- Scikit-Learn also has `StratifiedShuffleSplit()` that does stratified sampling.
- **Stratification** ensures that the test samples are representative of the target categories.

2.2.1. Create a Test Set: User-defined function

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

You can then use this function like this:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

2.2.2. Create a Test Set: Using Scikit-Learn functions

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Stratification is usually done on the target class.

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Outline

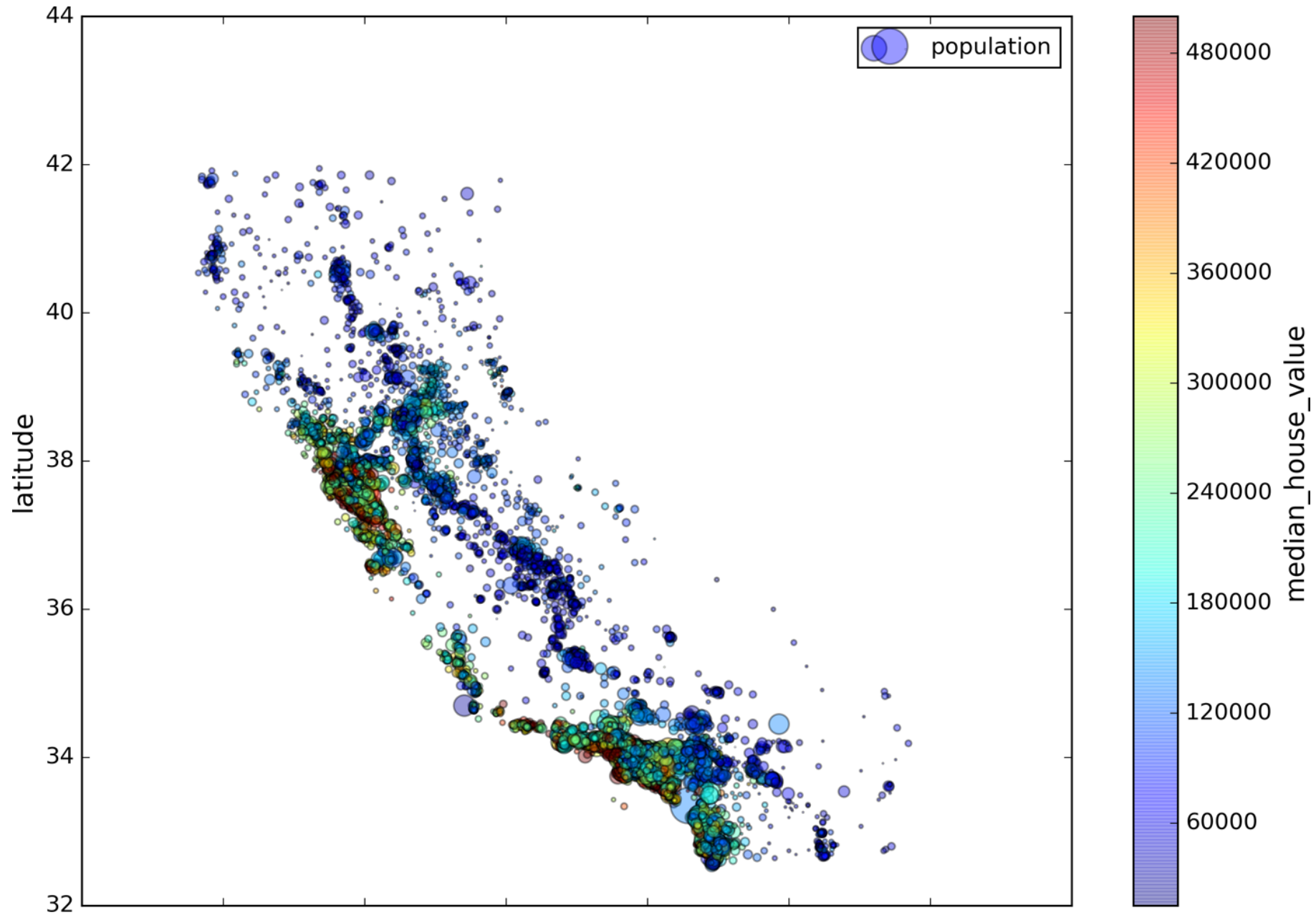
1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

3. Discover and Visualize the Data to Gain Insights

- **Visualize** geographical data using

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,  
             s=housing["population"]/100, label="population",  
             c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,  
             )  
plt.legend()
```

alpha: Transparency, **s**: size, **c**: color, **cmap**: blue to red



3.1. Looking for Correlations

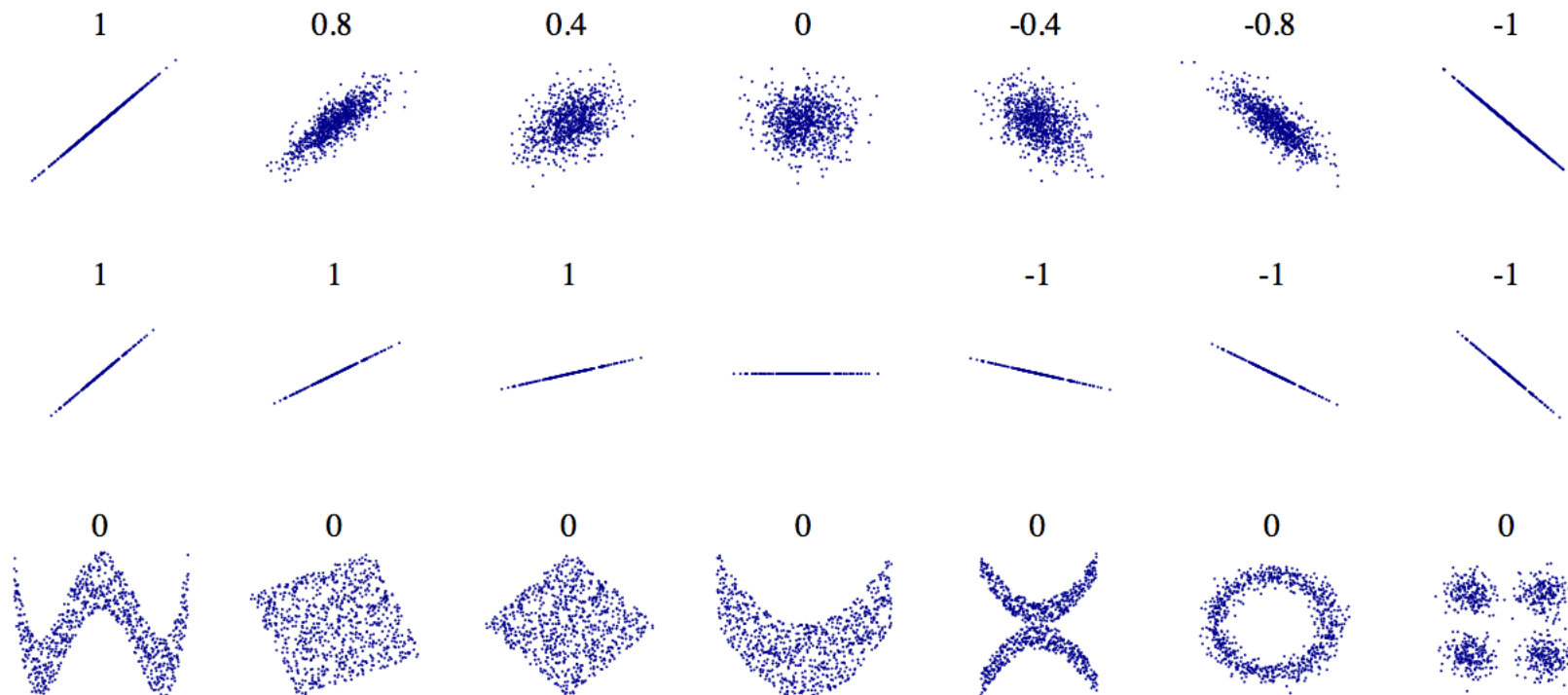
- Compute the **standard correlation coefficient** (also called **Pearson's r**) between every pair of attributes using **corr_matrix = housing.corr()**

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age   0.114220
households            0.064702
total_bedrooms        0.047865
population            -0.026699
longitude             -0.047279
latitude              -0.142826
```

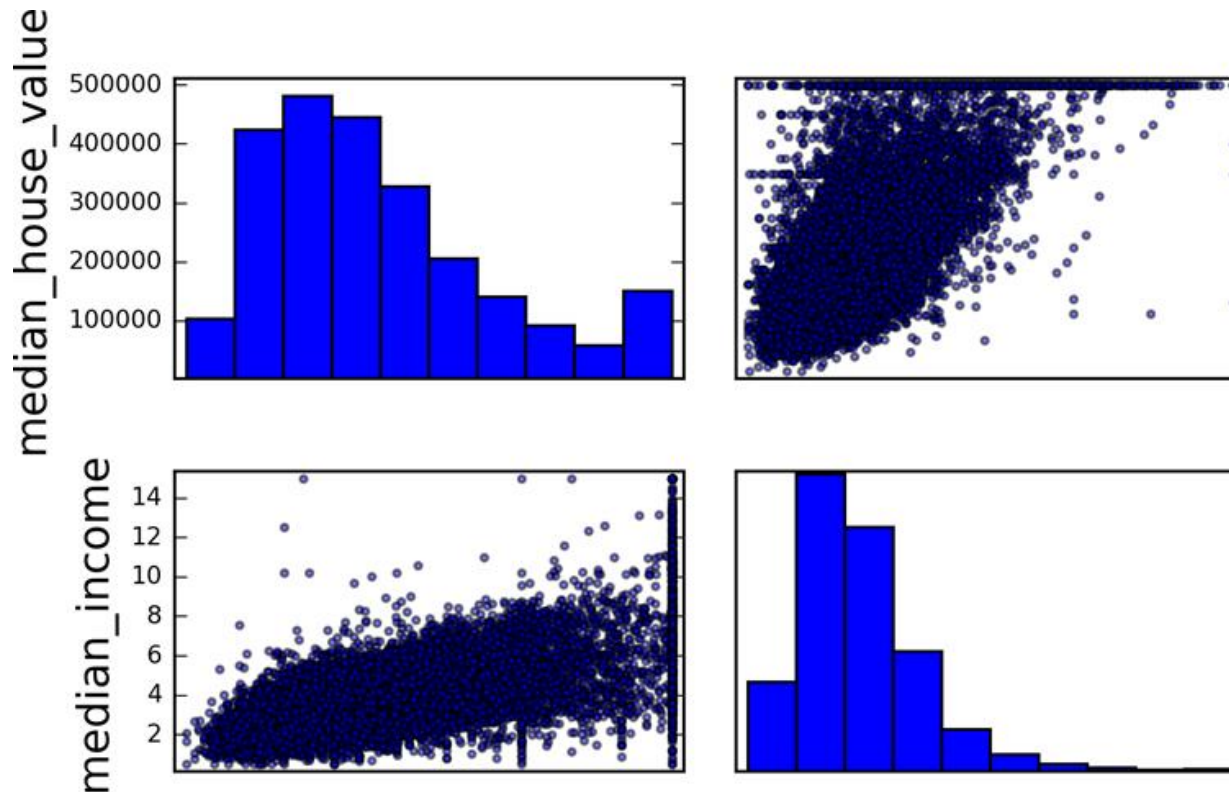
3.1. Looking for Correlations

- Zero linear correlation ($r = 0$) does not guarantee **independence**.



3.2. Pandas Scatter Matrix

```
from pandas.tools.plotting import scatter_matrix
attributes = ["median_house_value", "median_income"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```



3.3. Experimenting with Attribute Combinations

- Rooms per household is better than total rooms:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
```

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value      1.000000
median_income           0.687170
rooms_per_household    0.199343
total_rooms             0.135231
```

- Similarly, BMI is better than weight or height for medical purposes.

Outline

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

4. Prepare the Data for Machine Learning Algorithms

- **Split** to train and test (Done)
- **Separate** features from response
- Handle **missing** data
- Handle text and **categorical** features
- **Scale** (normalize) features
- Build preparation **pipeline**

4. Prepare the Data for Machine Learning Algorithms

- **Separate** the **features** from the **response**.

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

- **Options** of handling **missing features**:

1. **Get rid** of the corresponding **districts**
2. **Get rid** of the whole **attribute**
3. **Set the values** to some value (0, mean, median, etc.)

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)      # option 2
median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

4.1. Handling Missing Features Using Scikit-Learn

- Use **SimpleImputer** on the numerical features. Need to remove categorical variables before doing the fit. The attribute **statistics_** has the means.

```
from sklearn.preprocessing import SimpleImputer
imputer = SimpleImputer(strategy="median")
housing_num = housing.drop("ocean_proximity", axis=1)
imputer.fit(housing_num)
>>> imputer.statistics_
array([ -118.51 ,  34.26 ,  29. , 2119. ,  433. , 1164. ,  408. ,  3.5414])
>>> housing_num.median().values
array([ -118.51 ,  34.26 ,  29. , 2119. ,  433. , 1164. ,  408. ,  3.5414])
X = imputer.transform(housing_num)
```



NumPy array

4.2. Handling Text and Categorical Attributes

- **ocean_proximity** is categorical feature.

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
      ocean_proximity
17606      <1H OCEAN
18632      <1H OCEAN
14650      NEAR OCEAN
 3230           INLAND
 3555      <1H OCEAN
19480           INLAND
 8879      <1H OCEAN
13685           INLAND
 4937      <1H OCEAN
 4861      <1H OCEAN
```

4.2. Handling Text and Categorical Attributes

- Most machine learning algorithms prefer to work with numbers.

Converting to numbers:

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
```

```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

Numerical values
imply distances

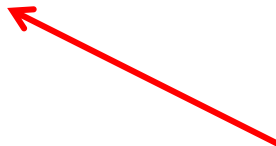


```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

4.2. Handling Text and Categorical Attributes

- To ensure encoding neutrality, we can use the one-hot encoding.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> cat_encoder = OneHotEncoder()
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
  with 16512 stored elements in Compressed Sparse Row format>
>>> housing_cat_1hot.toarray()
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```



Converts sparse matrix
to dense matrix.

4.3. Custom Transformers

- Scikit-Learn allows you to create your **own transformers**.
- You can create a transformer to create **derived features**.
- Create a class and implement three methods: **fit()** (returning self), **transform()**, and **fit_transform()**. Include base classes:
 - **TransformerMixin** to get **fit_transform()**
 - **BaseEstimator** to get **get_params()** and **set_params()**

4.3. Custom Transformers

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, household_ix = 3, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        return np.c_[X, rooms_per_household]

attr_adder = CombinedAttributesAdder()
housing_extra_attribs = attr_adder.transform(housing.values)
```

4.4. Feature Scaling

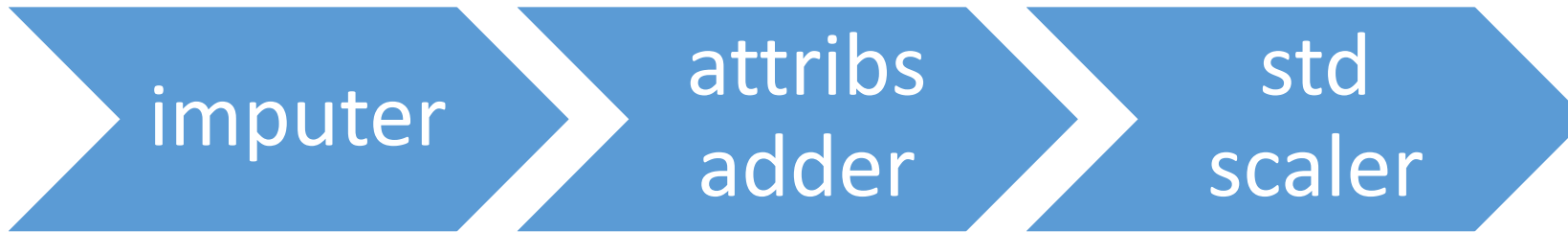
- ML algorithms generally **don't perform well** when the input numerical attributes have **very different scales**.
- Scaling techniques:
 - **Min-max scaling**

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- **Standardization**

$$x' = \frac{x - \bar{x}}{\sigma}$$

4.5. Transformation Pipelines



```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

4.6. Full Pipeline

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", OneHotEncoder(), cat_attribs),
])

housing_prepared = full_pipeline.fit_transform(housing)
```



Dense array

Outline

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

5. Select and Train a Model

- **Linear regressor**
- Using **RMSE** for evaluation
- **Decision tree regressor**
- **k-fold** cross validation
- **Random forests regressor**

5. Select and Train a Model

- Let us start by training a simple **linear regressor**.

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

- Try it out on five instances from the training set.

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:\t", lin_reg.predict(some_data_prepared))
Predictions:      [ 303104.   44800.  308928.  294208.  368704.]
>>> print("Labels:\t\t", list(some_labels))
Labels:           [359400.0, 69700.0, 302100.0, 301300.0, 351900.0]
```

50% off

5.1. Evaluate the Model on the Entire Training Set

- Use RMSE

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.413493824875
```

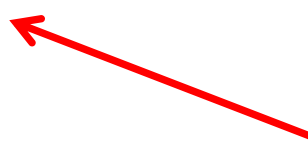
This is not a satisfactory result as the **median_housing_values** range between \$120,000 and \$265,000.

5.2. Try the Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)

>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```



Overfitting: It has memorized
the entire training set!

5.3. Better Evaluation Using Cross-Validation

- Segment the training data into **10 sets** and repeat training and evaluation 10 times.

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                        scoring="neg_mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)

>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [70194.33680785 66855.16363941 ... ]
Mean: 71407.68766037929 ←
Standard deviation: 2439.4345041191004
```

Worse than Linear
Regressor

5.4. Try the Random Forests Regressor

- Repeating training and evaluation:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
18603.515021376355
>>> display_scores(forest_rmse_scores)
Scores: [49519.80364233 47461.9115823 50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574 47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096 ← Best Accuracy
Standard deviation: 2097.0810550985693
```

Outline

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

6. Fine-Tune Your Model

- Fine-tune your system by fiddling with:
 - The hyperparameters
 - Removing and adding features
 - Changing feature preprocessing techniques
- Can experiment manually. But it is best to automate this process using Scikit-Learn:
 - **GridSearchCV**
 - or **RandomizedSearchCV**

6.1. Grid Search

- Can automate exploring a search space of $3 \times 4 + 2 \times 3 = 12 + 6 = 18$

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

6.2 Examine the Results of Your Grid Search

- Can examine the best hyperparameters using:

```
>>> grid_search.best_params_  
{'max_features': 8, 'n_estimators': 30}
```

- Can examine all search results using:

```
>>> cvres = grid_search.cv_results_  
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):  
...     print(np.sqrt(-mean_score), params)  
...  
63669.05791727153 {'max_features': 2, 'n_estimators': 3}  
55627.16171305252 {'max_features': 2, 'n_estimators': 10}  
...  
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
```



Best Tuned Accuracy

6.2 Evaluate Your System on the Test Set

- The final model is the best estimator found by the grid search.
- To evaluate it on the test set, transform the test features, predict using transformed features, and evaluate accuracy.

```
final_model = grid_search.best_estimator_  
X_test = strat_test_set.drop("median_house_value", axis=1)  
y_test = strat_test_set["median_house_value"].copy()  
X_test_prepared = full_pipeline.transform(X_test)  
final_predictions = final_model.predict(X_test_prepared)  
final_mse = mean_squared_error(y_test, final_predictions)  
final_rmse = np.sqrt(final_mse) # => evaluates to 48,209.6
```

Better than train set!



6.3 Save Your Best Model for the Production System

```
from sklearn.externals import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

Outline

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

7. Present Your Solution

- Present your solution highlighting:
 - What you have learned
 - What worked and what did not
 - What assumptions were made
 - What your system's limitations are
- Document everything, and create nice presentations with:
 - Clear visualizations
 - Easy-to-remember statements, e.g., “the median income is the number one predictor of housing prices”.

8. Launch, Monitor, and Maintain Your System

- Prepare your production program that uses your best trained model and launch it.
- Monitor the accuracy of your system. Also monitor the input data.
- Retrain your system periodically using fresh data.

Summary

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Select a model and train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor, and maintain your system
9. Exercises

Exercise

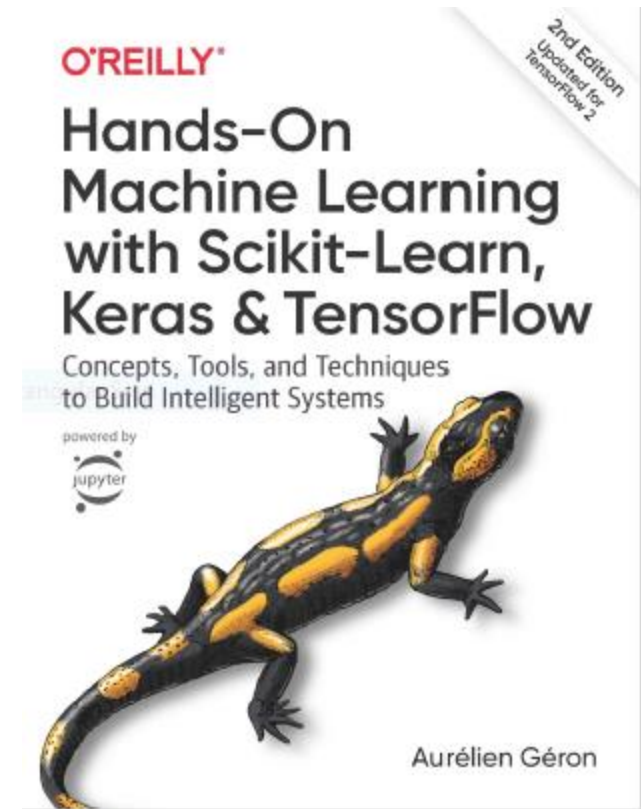
- Try a Support Vector Machine regressor (`sklearn.svm.SVR`), with various hyperparameters such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Don't worry about what these hyperparameters mean for now. How does the best SVR predictor perform?

Classification

Prof. Gheith Abandah

Reference

- Chapter 3: **Classification**



- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>

Introduction

- YouTube Video: **Machine Learning - Supervised Learning Classification** from Cognitive Class

<https://youtu.be/Lf2bCQIktTo>

Outline

1. MNIST dataset
2. Training a binary classifier
3. Performance measures
4. Multiclass classification
5. Multilabel classification
6. Exercise

1. MNIST Dataset

- **MNIST** is a set of 70,000 small images of **handwritten digits**.
- Available from mldata.org
- **Scikit-Learn** provides **download** functions.



1.1. Get the Data

```
>>> from sklearn.datasets import fetch_openml
>>> mnist = fetch_openml('mnist_784', version=1)
>>> mnist.keys()
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',
           'categories', 'url'])
```


1.2. Extract Features and Labels

```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

There are 70,000 images, and each image has **784** features.

This is because each image is **28×28** pixels, and each feature simply represents one pixel's intensity, from **0 (white)** to **255 (black)**.

1.3. Examine One Image

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
```

```
plt.imshow(some_digit_image, cmap = mpl.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```

```
>>> y[0]
'5'
```



1.4. Split the Data

- The MNIST dataset is actually already split into a **training set** (the first 60,000 images) and a **test set** (the last 10,000 images).
- The training set is **already shuffled**.

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Outline

1. MNIST dataset
2. Training a binary classifier
3. Performance measures
4. Multiclass classification
5. Multilabel classification
6. Exercise

2. Training a Binary Classifier

- A binary classifier can classify **two classes**.
- For example, classifier for the number 5, capable of distinguishing between two classes, **5** and **not-5**.

```
y_train_5 = (y_train == 5)  
y_test_5 = (y_test == 5)
```

True for all 5s, False for all other digits.

```
from sklearn.linear_model import SGDClassifier
```

```
sgd_clf = SGDClassifier(random_state=42)  
sgd_clf.fit(X_train, y_train_5)
```

Stochastic Gradient Descent (SGD) classifier

```
>>> sgd_clf.predict([some_digit])  
array([ True])
```

Outline

1. MNIST dataset
2. Training a binary classifier
3. Performance measures
4. Multiclass classification
5. Multilabel classification
6. Exercise

3. Performance Measures

- **Accuracy**: Ratio of correct predictions
- **Confusion matrix**
- **Precision** and **recall**
- **F1 Score**
- **Precision/recall tradeoff**

3.1. Accuracy

```
y_pred = clone_clf.predict(X_test_fold)
n_correct = sum(y_pred == y_test_fold)
print(n_correct / len(y_pred))
```

Example how to find the accuracy.

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.96355, 0.93795, 0.95615])
```

Using the `cross_val_score()` function to find the accuracy on three folds

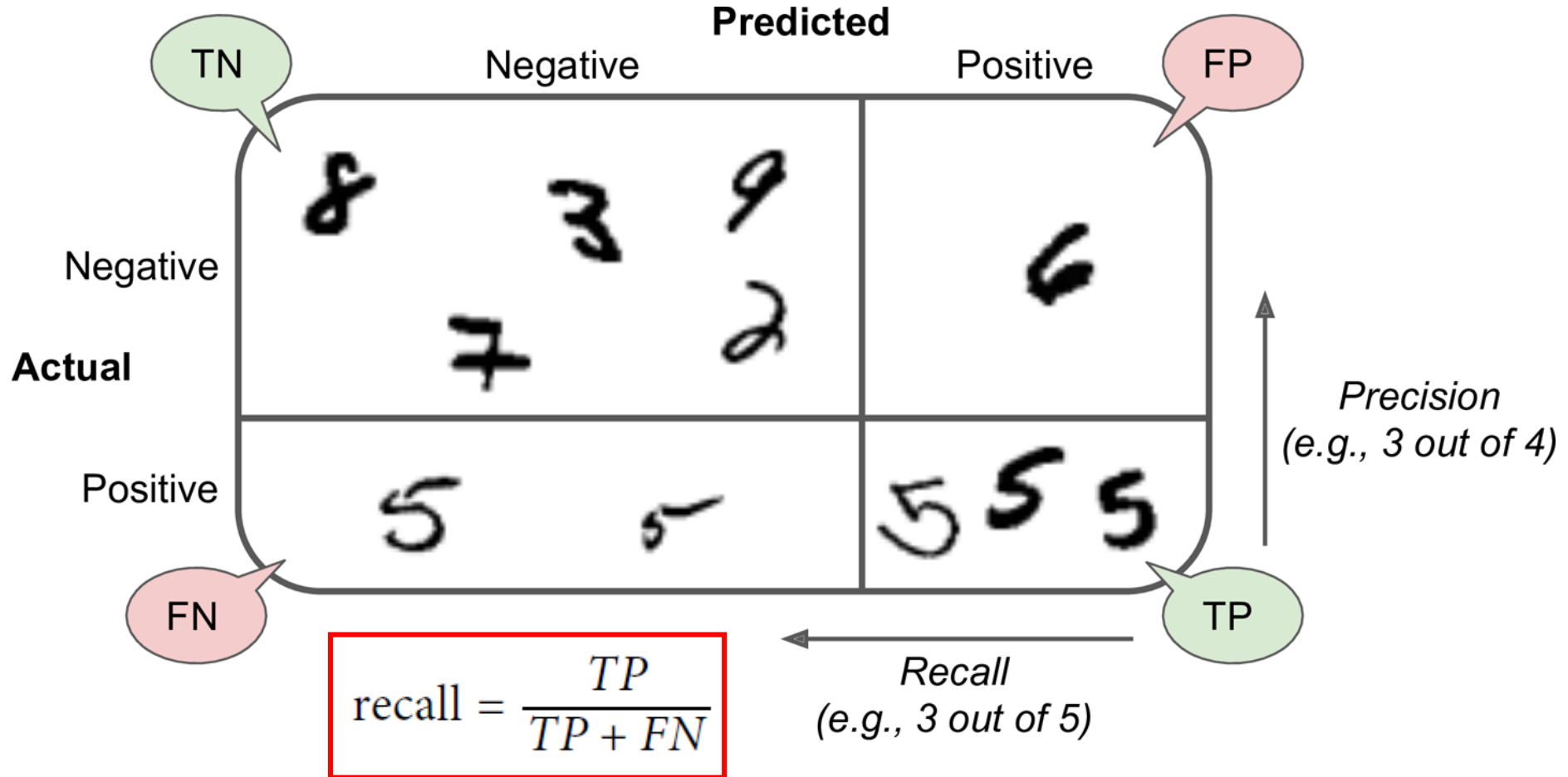
3.1. Accuracy

- Use `cross_val_predict()` to predict the targets of the entire training set.

```
from sklearn.model_selection import cross_val_predict  
  
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

3.2. Confusion Matrix

$$\text{precision} = \frac{TP}{TP + FP}$$



3.2. Confusion Matrix

- Scikit Learn has a function for finding the **confusion matrix**.

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53057, 1522],
       [ 1325, 4096]])
```

- The first row is for the non-5s (the **negative** class):
 - 53,057 correctly classified (**true negatives**)
 - 1,522 wrongly classified (**false positives**)
- The second row is for the 5s (the **positive** class):
 - 1,325 wrongly classified (**false negatives**)
 - 4,096 correctly classified (**true positives**)

3.3. Precision and Recall

Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

The precision and recall are smaller than the accuracy.
Why?

3.4. F1 Score

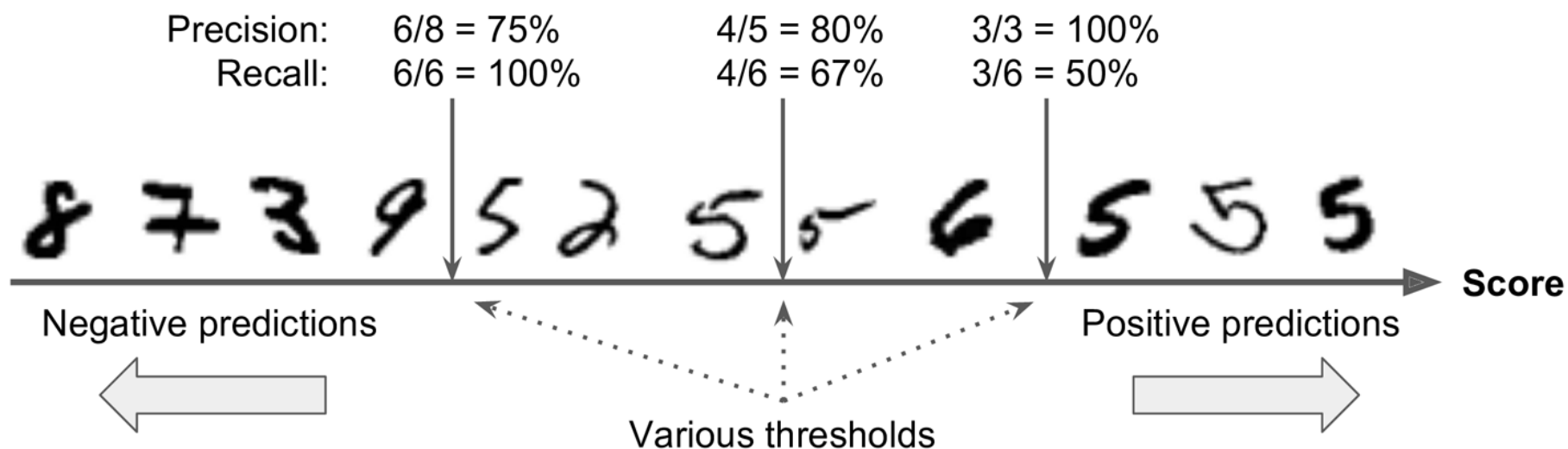
- The F1 Score combines the precision and recall in one metric (**harmonic mean**).

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

3.5. Precision/Recall Tradeoff

- **Increase** the **decision threshold** to improve the precision when it is **bad** to have FP.
- **Decrease** the decision threshold to improve the recall when it is **important not to miss FN**.



3.5. Precision/Recall Tradeoff

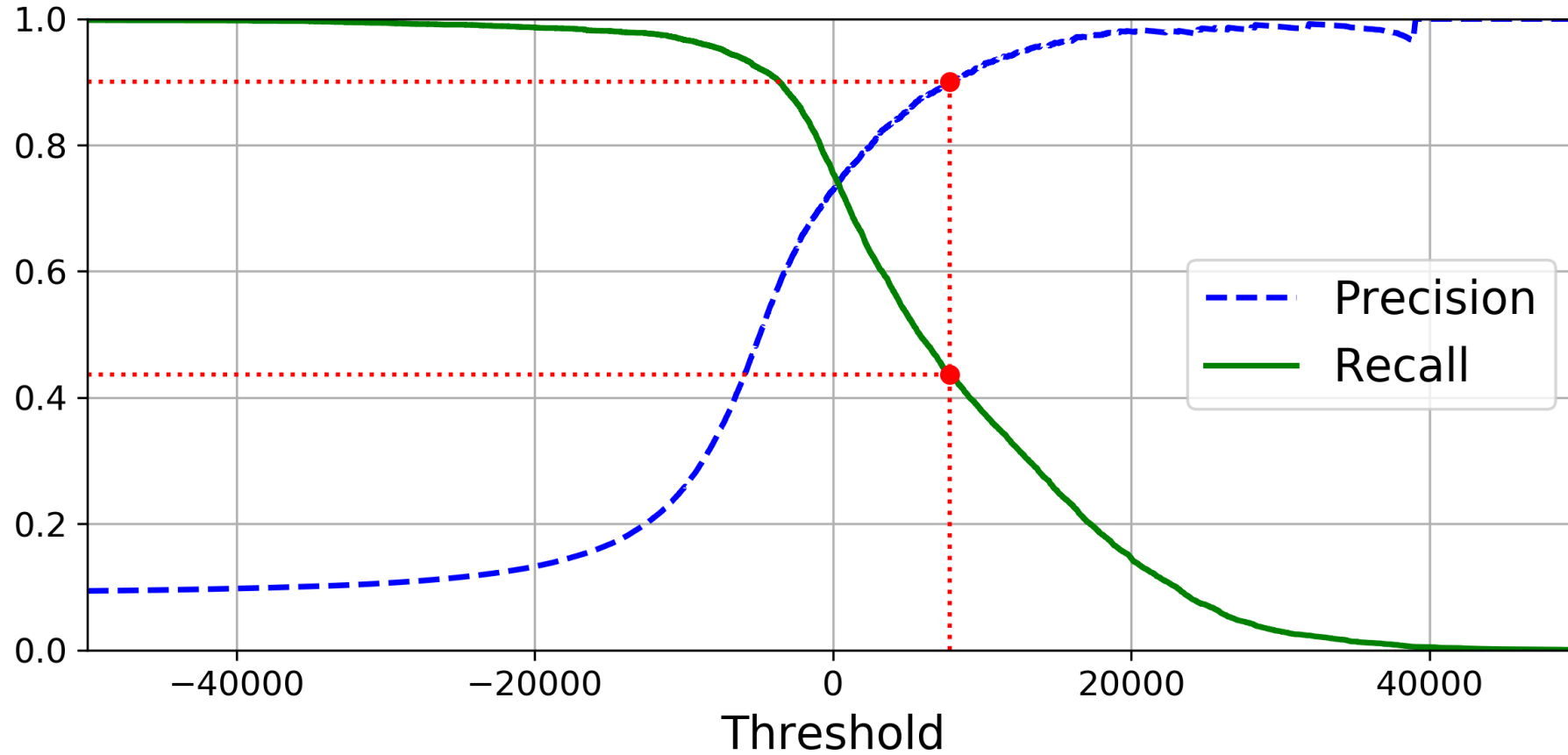
- The function `cross_val_predict()` can return **decision scores** instead of predictions.

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,  
                             method="decision_function")
```

- These scores can be used to compute precision and recall for all possible thresholds using the **precision_recall_curve()** function.

```
from sklearn.metrics import precision_recall_curve  
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

3.5. Precision/Recall Tradeoff



3.5. Precision/Recall Tradeoff

- For **larger precision**, **increase the threshold**, and **decrease it** for **larger recall**.
- **Example**: To get 90% precision.

The first threshold with precision $\geq 90\%$

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)] # ~7816
y_train_pred_90 = (y_scores >= threshold_90_precision)
>>> precision_score(y_train_5, y_train_pred_90)
0.9000380083618396
>>> recall_score(y_train_5, y_train_pred_90)
0.4368197749492714
```

True when score
 \geq new threshold

Outline

1. MNIST dataset
2. Training a binary classifier
3. Performance measures
4. Multiclass classification
5. Multilabel classification
6. Exercise

4. Multiclass Classification

- Multiclass classifiers can distinguish between **more than two classes**.
- Some **algorithms** (such as Random Forest classifiers or Naive Bayes classifiers) are **capable of handling multiple classes** directly.
- **Others** (such as Support Vector Machine classifiers or Linear classifiers) are **strictly binary classifiers**.
- There are **two main strategies** to perform multiclass classification using multiple binary classifiers.

4.1. One-versus-All (OvA) Strategy

- For example, classify the digit images into 10 classes (from 0 to 9) to **train 10 binary classifiers**, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on).
- Then to classify an image, get the decision score from each classifier for that image and select the class whose classifier outputs the **highest score**.

4.2. One-versus-One (OvO) Strategy

- Train a binary classifier **for every pair** of digits.
- If there are N classes, need $N \times (N - 1) / 2$ classifiers. For MNIST, **need 45 classifiers**.
- To classify an image, run the image through all 45 classifiers and see which class **wins the most duels**.
- The main advantage of **OvO** is that each classifier only needs to be **trained on a subset** of the training set.
- OvO is preferred for algorithms (such as **Support Vector Machine**) that scale poorly with the size of the training set.

4.3. Scikit Learn Support of Multiclass Classification

- **Scikit-Learn** detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs **OvA** (except for **SVM** classifiers for which it uses **OvO**).

```
>>> sgd_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)
```

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=42)
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([5], dtype=uint8)
```

Better
classifier than
SGD

4.3. Scikit Learn Support of Multiclass Classification

- Note that the multiclass task is harder than the binary task.
- **Binary task**

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.96355, 0.93795, 0.95615])
```

- **Multiclass task**

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8489802 , 0.87129356, 0.86988048])
```

4.4. Error Analysis

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [    0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

Many images are misclassified as 8s.

Outline

1. MNIST dataset
2. Training a binary classifier
3. Performance measures
4. Multiclass classification
5. **Multilabel classification**
6. Exercise

5. Multilabel Classification

- Classifiers that output **multiple classes for each instance**.

```
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]
```

```
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

Popular algorithm

```
>>> knn_clf.predict([some_digit])
array([[False, True]], dtype=bool)
```

Summary

1. MNIST dataset
2. Training a binary classifier
3. Performance measures
4. Multiclass classification
5. Multilabel classification
6. Exercise

Exercise

- Try to build a classifier for the MNIST dataset that achieves over 97% accuracy on the test set. Hint: the **KNeighborsClassifier** works quite well for this task; you just need to find good hyperparameter values (try a grid search on the **weights** and **n_neighbors** hyperparameters).

Machine Learning

Project

General

- To enable the students to get hands-on experience in the design, implementation and evaluation of machine learning systems.
- Teams: One students each
- Solve a practical machine learning problem of your choice.
- Use Python.
- Good projects involve using alternative approaches and evaluating their performance in solving the problem.

General

- Article: **Project-Based Learning for Data Scientists: Becoming a Data Scientist just became a whole lot easier**
- Author: Kishen Sharma
- Link: <https://towardsdatascience.com/project-based-learning-for-data-scientists-df6a8f74e4a1>

General

- **Marks:**

- Report 50%
- Presentation 50%

- **Timing:**

- Mon 3 May, 2021 Submit proposal
- Wed 26 May, 2021 Submit report & present project

Project Proposal

- One to two-page proposal
- Specify problem
- Specify sample size and source
- Structure
 - Title
 - Student name
 - Problem definition
 - Data description
 - Samples

Research Report

- Four to 8-page report
- Use IEEE A4 conference template at http://www.ieee.org/conferences_events/conferences/publishing/templates.html
- In the introduction, include
 - Motivation
 - Problem definition
 - Literature review
- Describe your data and development environment.
- Describe any preprocessing, feature extraction and selection, techniques used, and post-processing.
- Give results and comments
- Give conclusions (work done, main results, future work)
- Include your source code in an appendix after the list of references.

Research Presentation

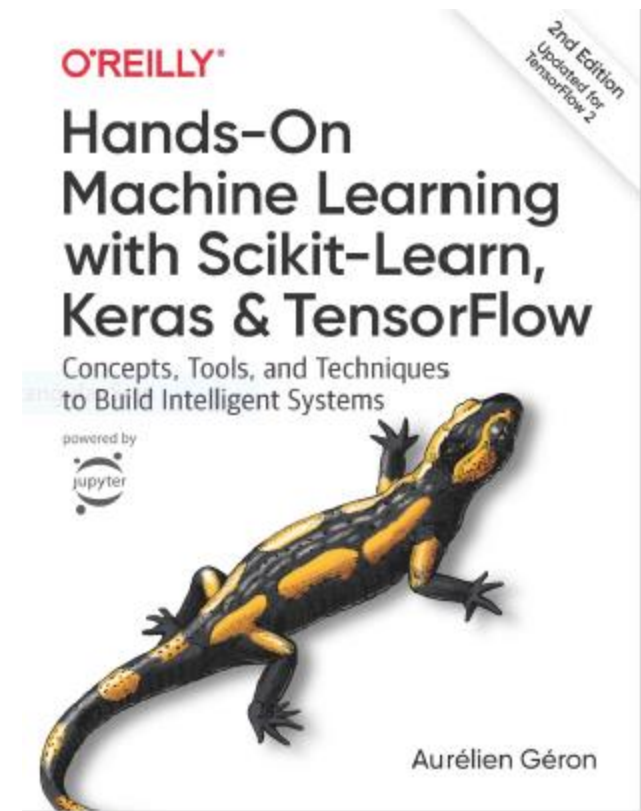
- Prepare Power Point slides
- Ten minutes long
- Must be clear and useful presentation, must add knowledge to fellow colleagues

Training Models and Regression

Prof. Gheith Abandah

Reference

- Chapter 4: **Training Models**



- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>

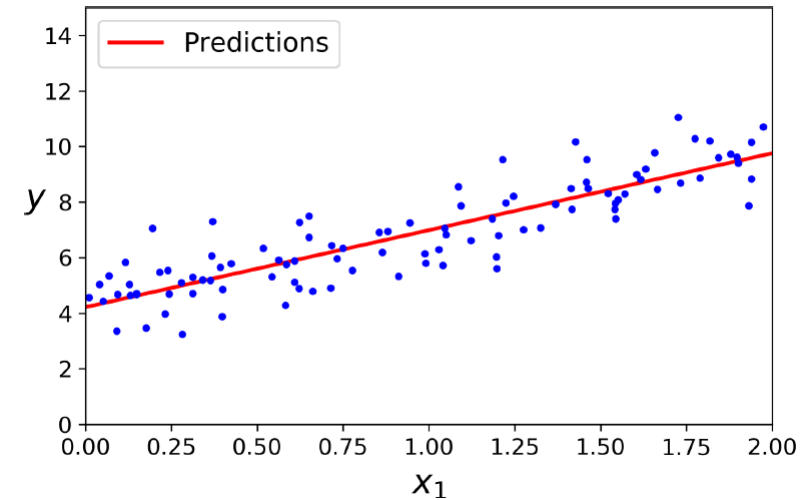
Outline

1. Linear Regression
2. Gradient Descent
3. Gradient Descent Variants
 1. Batch Gradient Descent
 2. Stochastic Gradient Descent
 3. Mini-batch Gradient Descent
4. Learning Curves
5. Early Stopping
6. Exercises

Linear Regression

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- \hat{y} is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).



$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

Analytical Solution

- The Root Mean Square Error (RMSE) is used as **cost function**.

$$\text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2$$

- Minimizing this cost gives the following solution (**normal function**):

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad \leftarrow \text{Complexity } \mathcal{O}(mn^2)$$

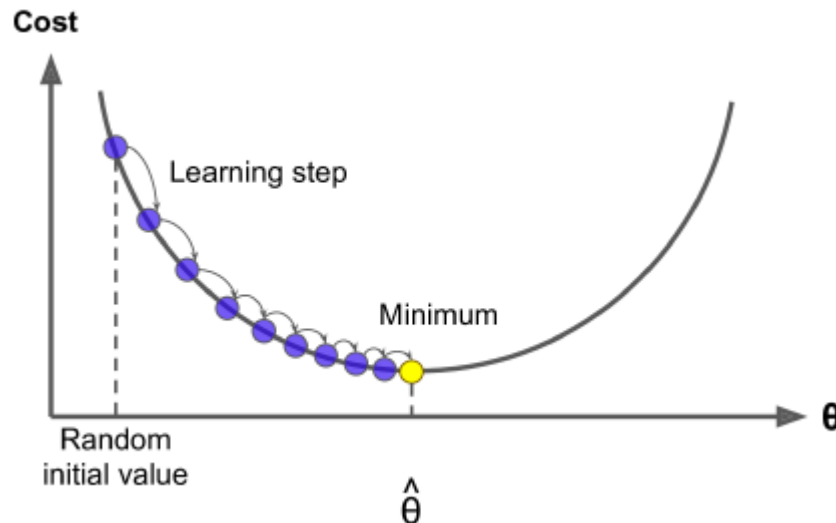
- $\hat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ that minimizes the cost function.
- \mathbf{y} is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Outline

1. Linear Regression
2. Gradient Descent
3. Gradient Descent Variants
 1. Batch Gradient Descent
 2. Stochastic Gradient Descent
 3. Mini-batch Gradient Descent
4. Learning Curves
5. Early Stopping
6. Exercises

Gradient Descent

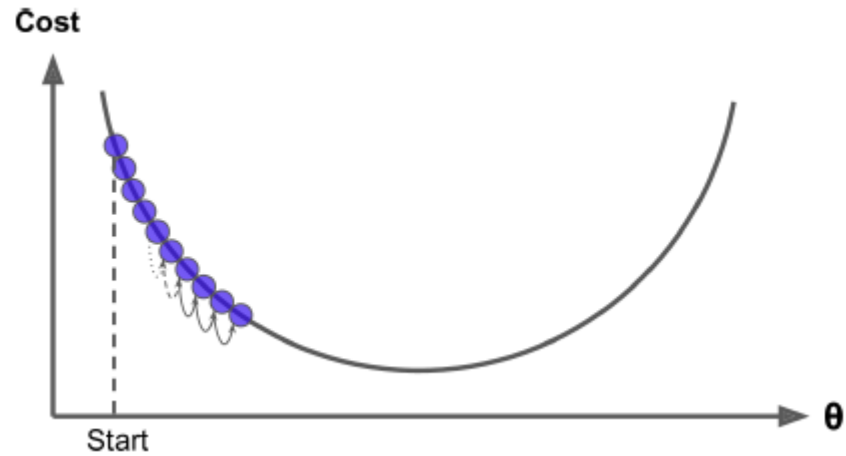
- **Generic optimization algorithm** capable of finding optimal solutions to a wide range of problems.
- **Tweaks parameters** iteratively in order **to minimize a cost function**.



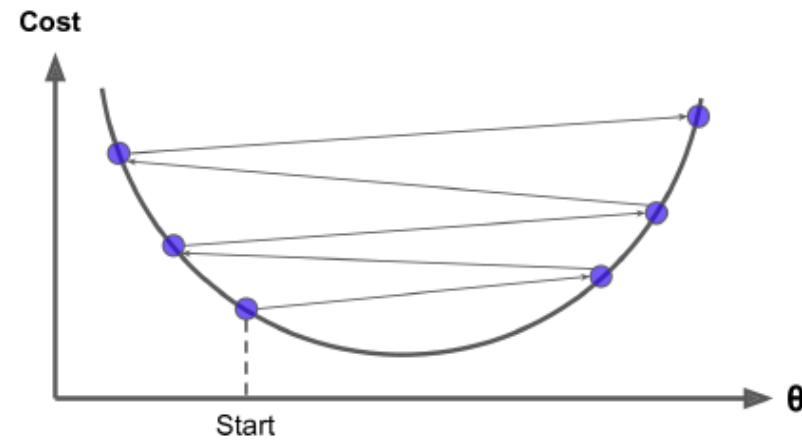
$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Learning Rate

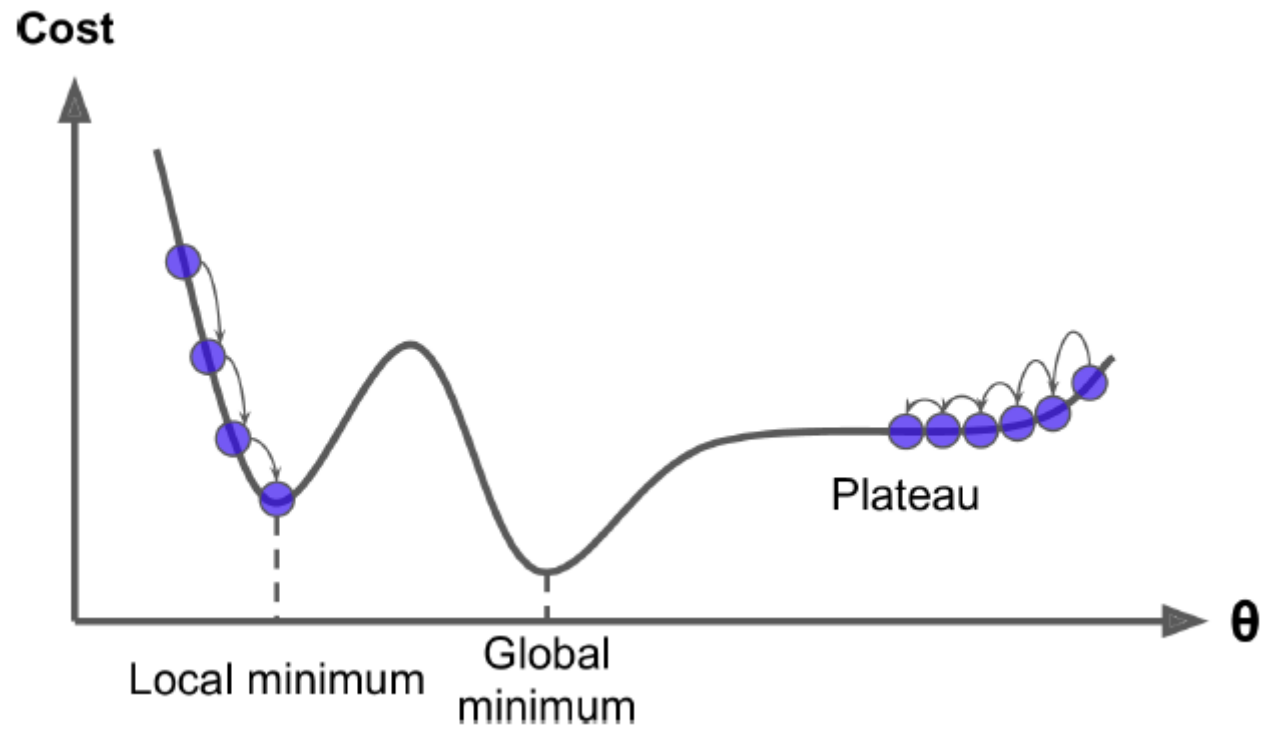
Too Small



Too Large

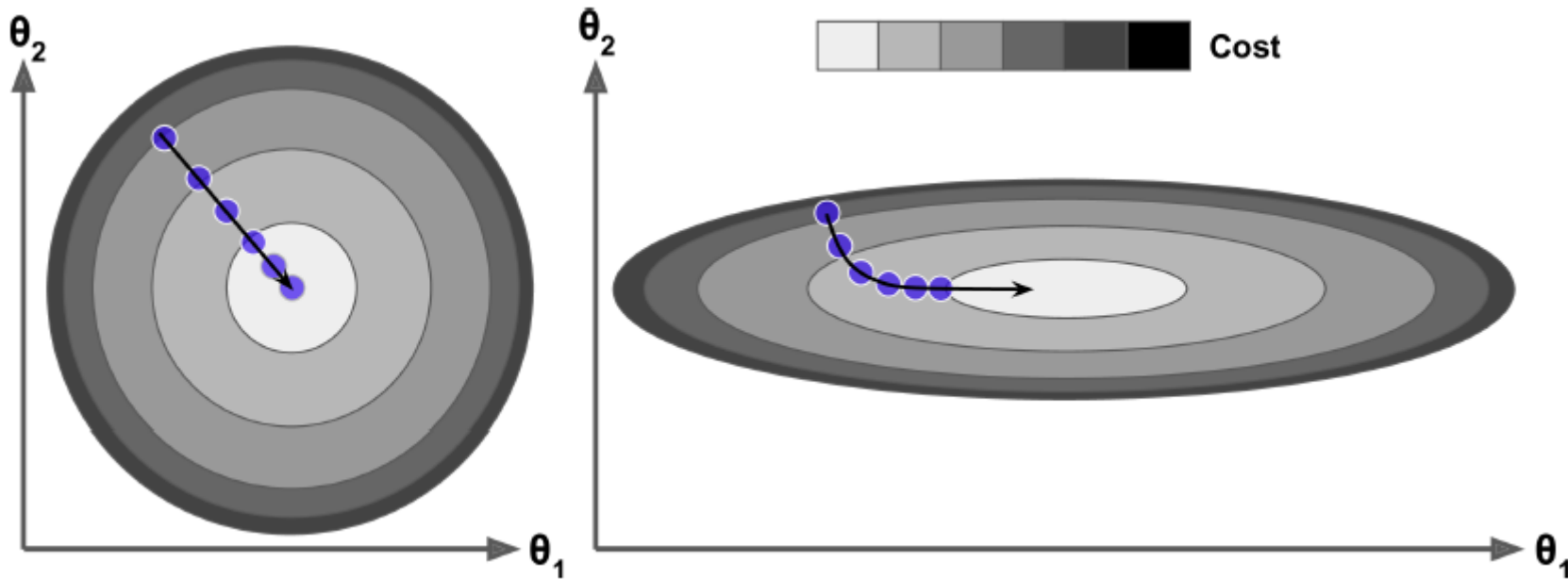


Gradient Descent Pitfalls



Feature Scaling

- Ensure that all features have a similar scale (e.g., using Scikit-Learn's **StandardScaler** class).
- Gradient Descent with and without feature scaling.



Outline

1. Linear Regression
2. Gradient Descent
3. Gradient Descent Variants
 1. Batch Gradient Descent
 2. Stochastic Gradient Descent
 3. Mini-batch Gradient Descent
4. Learning Curves
5. Early Stopping
6. Exercises

Batch Gradient Descent

- **Partial derivatives** of the cost function in θ_j

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

- **Gradient vector** of the cost function

$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

The entire training
Batch

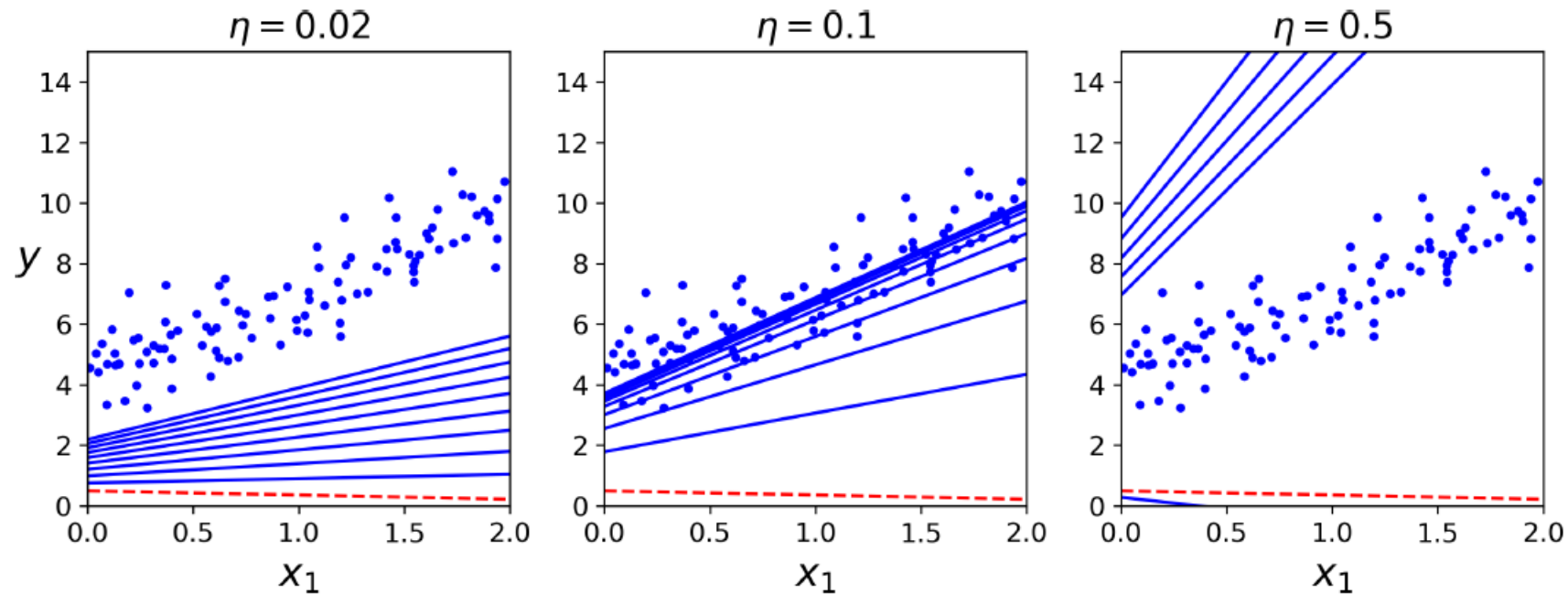
$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$$

Batch Gradient Descent

- Gradient Descent step

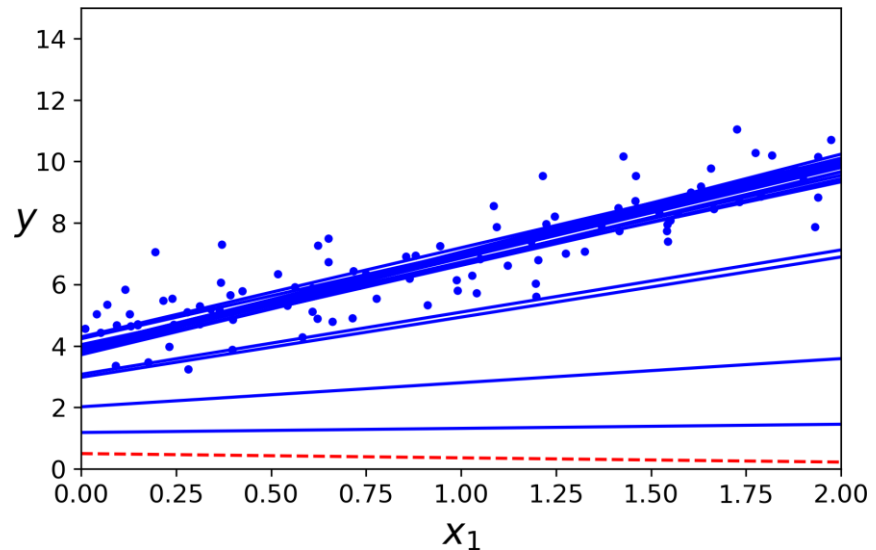
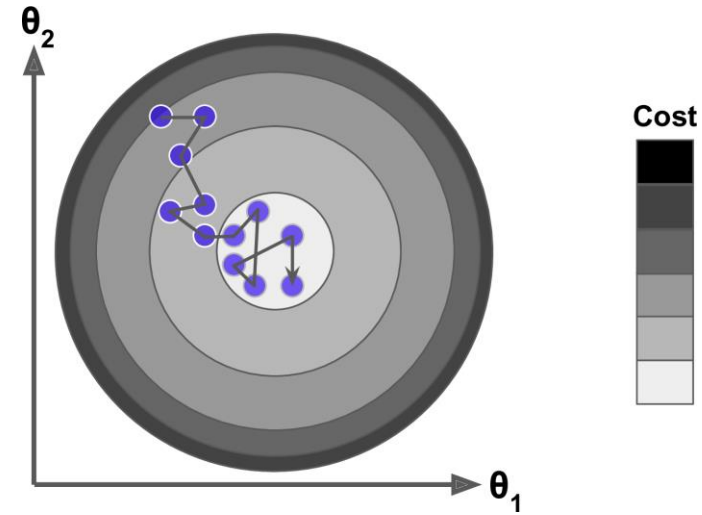
$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

- Gradient Descent with various learning rates



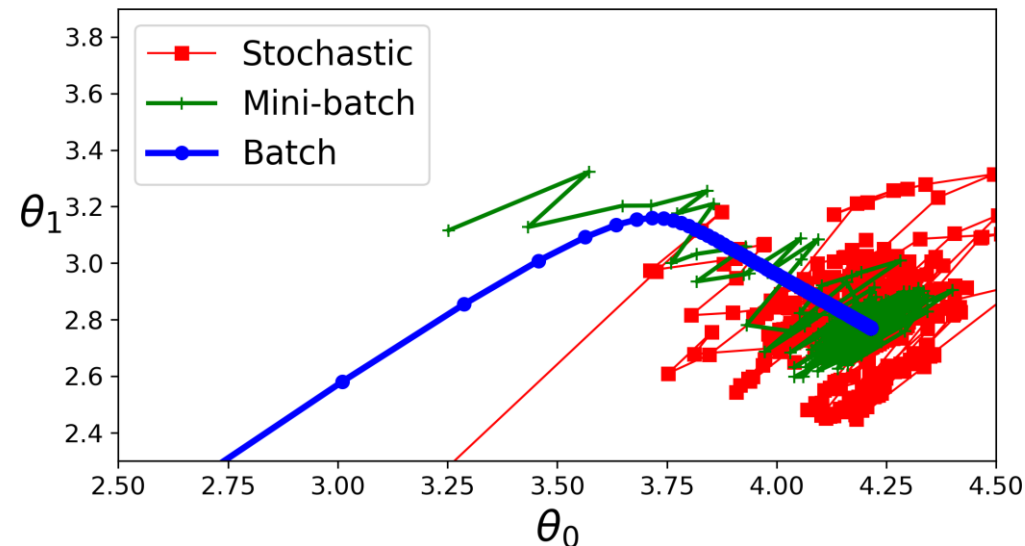
Stochastic Gradient Descent

- SGD **picks a random instance** in the training set at every step and computes the gradients.
- SGD is **faster** when the training set is large.
- Is **bouncy**
- Eventually gives **good solution**
- Can **escape local minima**



Mini-batch Gradient Descent

- Computes the gradients on small random sets of instances called **mini batches**.
- Benefits from **hardware accelerators** (e.g., GPU).
- **Less bouncy, better solution, escapes some local minima**

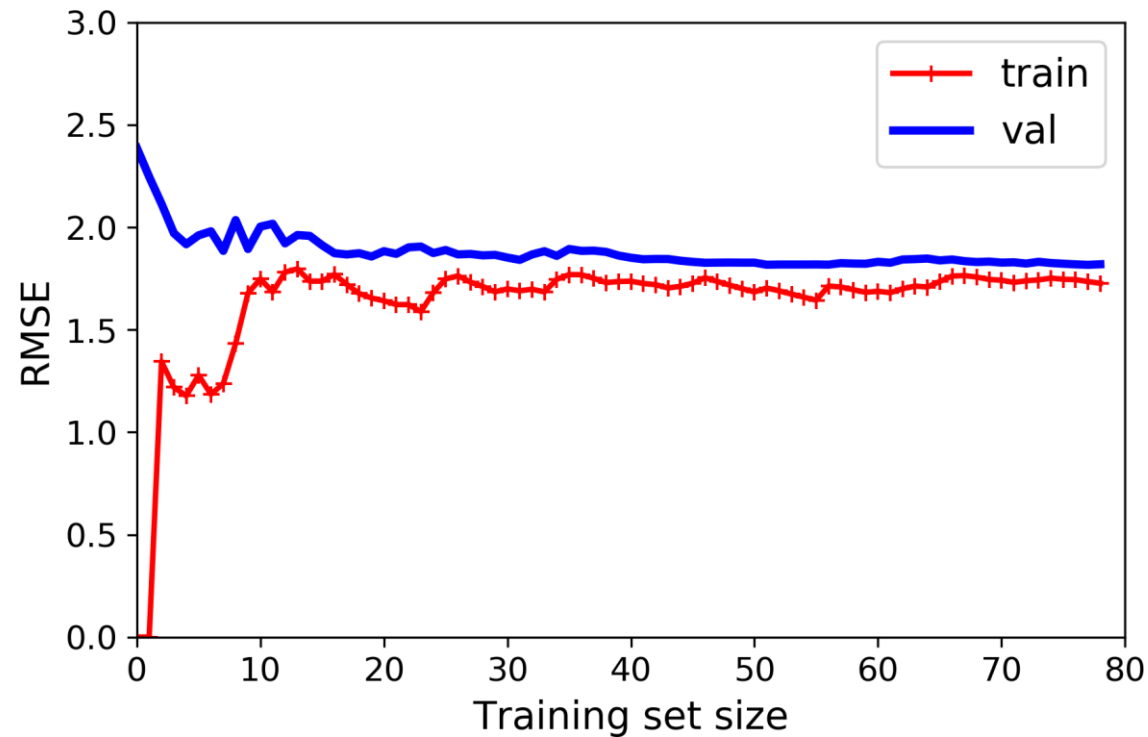


Outline

1. Linear Regression
2. Gradient Descent
3. Gradient Descent Variants
 1. Batch Gradient Descent
 2. Stochastic Gradient Descent
 3. Mini-batch Gradient Descent
4. Learning Curves
5. Early Stopping
6. Exercises

Learning Curves

- The **accuracy** on the **validation set** generally **increases** as the **training set** size increases.
- **Overfitting decreases** with larger training set.

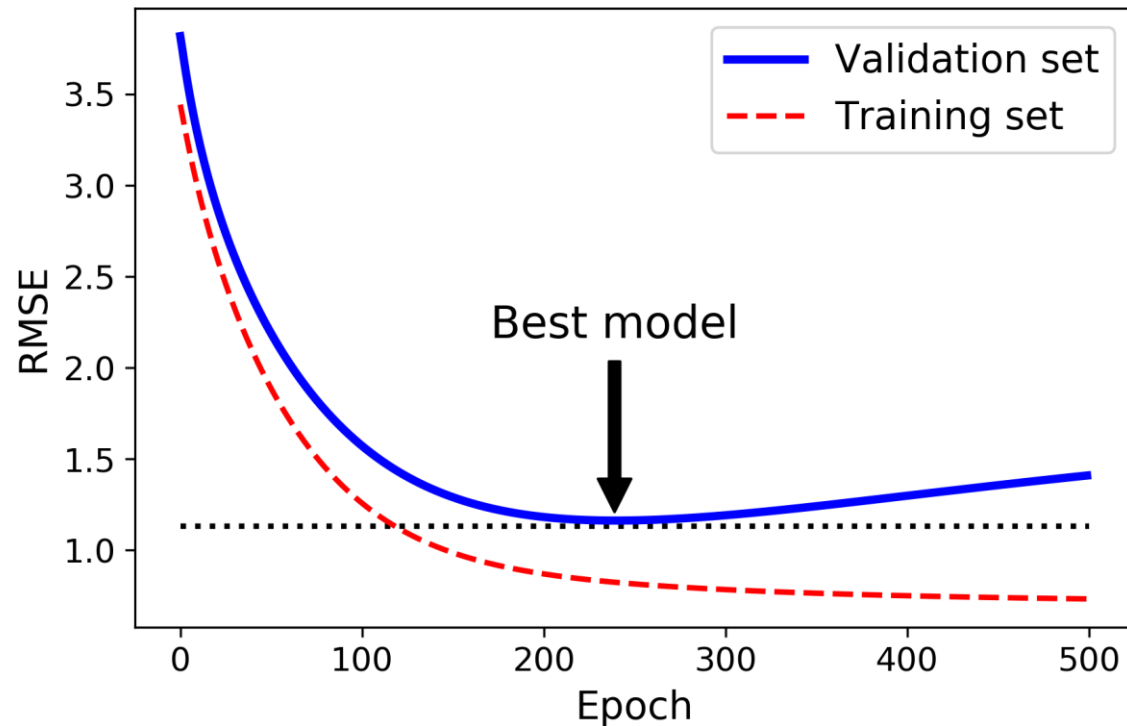


Outline

1. Linear Regression
2. Gradient Descent
3. Gradient Descent Variants
 1. Batch Gradient Descent
 2. Stochastic Gradient Descent
 3. Mini-batch Gradient Descent
4. Learning Curves
5. Early Stopping
6. Exercises

Early Stopping

- **Stop** training when the **validation error reaches a minimum**.
- Need to **save the best model**.



Outline

1. Linear Regression
2. Gradient Descent
3. Gradient Descent Variants
 1. Batch Gradient Descent
 2. Stochastic Gradient Descent
 3. Mini-batch Gradient Descent
4. Learning Curves
5. Early Stopping
6. Exercises

Exercises

1. What Linear Regression training algorithm can you use if you have a training set with millions of features?
2. Suppose the features in your training set have very different scales. What algorithms might suffer from this, and how? What can you do about it?
3. Do all Gradient Descent algorithms lead to the same model provided you let them run long enough?

Summary

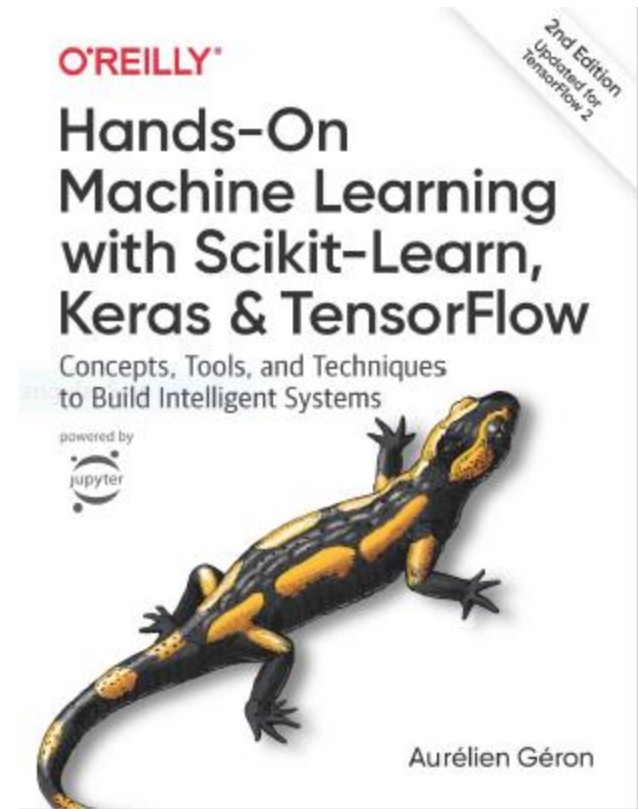
1. Linear Regression
2. Gradient Descent
3. Gradient Descent Variants
 1. Batch Gradient Descent
 2. Stochastic Gradient Descent
 3. Mini-batch Gradient Descent
4. Learning Curves
5. Early Stopping
6. Exercises

Classical Techniques

Prof. Gheith Abandah

Reference

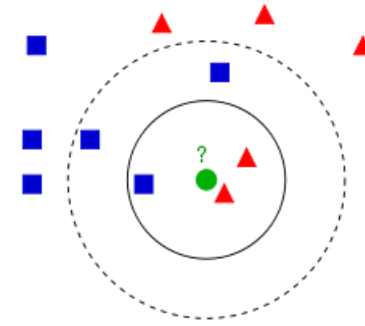
- Chapter 5: **Support Vector Machines**
 - Chapter 6: **Decision Trees**
 - Chapter 7: **Ensemble Learning and Random Forests**
-
- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>



Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

k-Nearest Neighbors



- Find a predefined number of training samples (k) closest in distance to the new point and predict the label from them: **regression** or **classification**.
- The number of samples can be a user-defined constant (**k-nearest neighbor learning**), or vary based on the local density of points (**radius-based neighbor learning**).
- The distance can be any metric measure: standard **Euclidean distance** is the most common choice.
- Reference: <https://scikit-learn.org/stable/modules/neighbors.html>

Nearest Neighbors Classification

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5,  
                                           weights='uniform', ... )
```

- **weights** can be: **uniform**: All points in each neighborhood are weighted equally, and **distance**: Weight points by the inverse of their distance.

- Example:

```
from sklearn.neighbors import KNeighborsClassifier  
knn_clf = KNeighborsClassifier()  
knn_clf.fit(X_train, y_train)
```

Nearest Neighbors Regression

```
class sklearn.neighbors.KNeighborsRegressor(n_neighbors=5,  
                                           weights='uniform', ... )
```

- The label assigned to a query point is computed based on the mean of the labels of its nearest neighbors.
- Example:

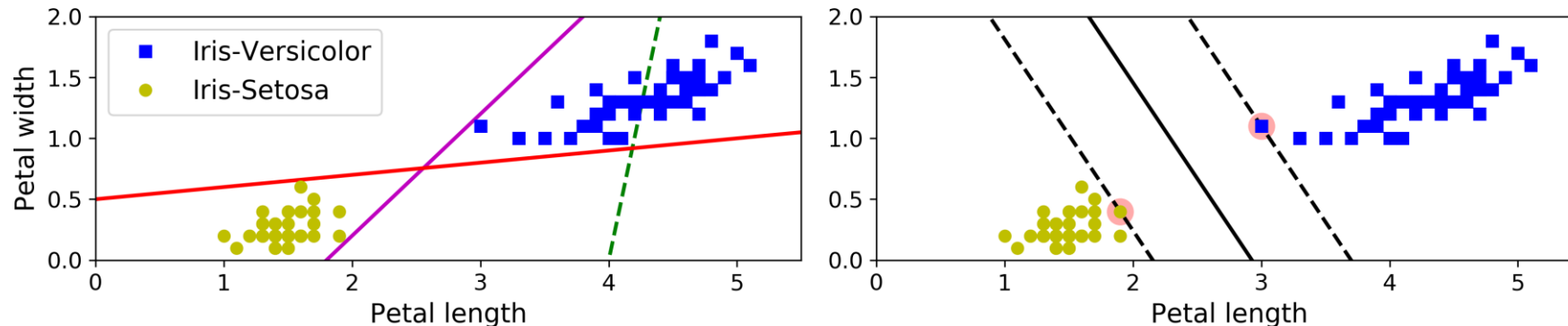
```
from sklearn.neighbors import KNeighborsRegressor  
model = KNeighborsRegressor(n_neighbors=3)  
model.fit(X, y)
```

Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

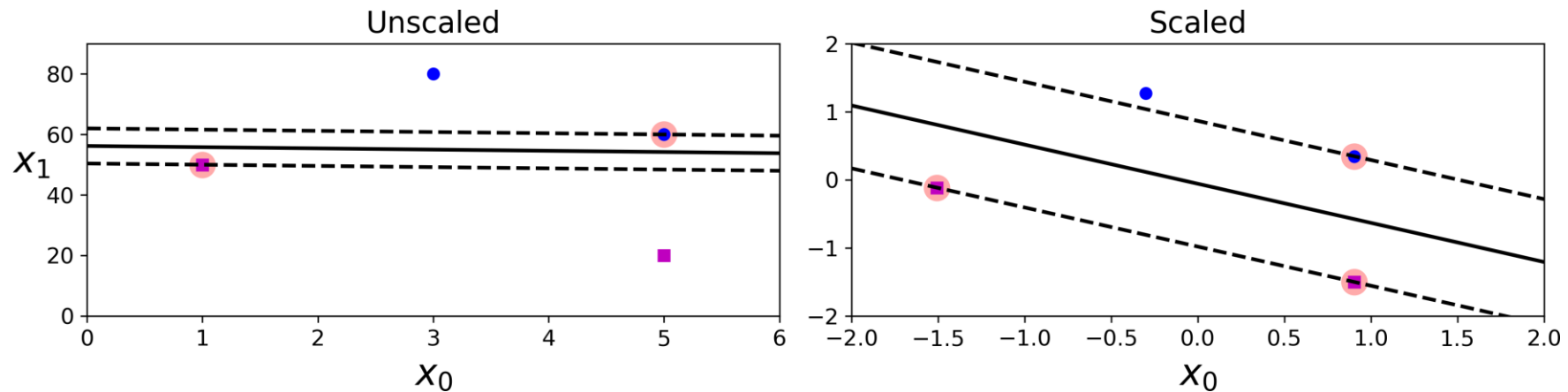
Support Vector Machine (SVM)

- Very **powerful** and **versatile** Machine Learning model, capable of performing **linear** or **nonlinear classification**, **regression**, and outlier detection.
- Well suited for classification of **complex** but **small-** or **medium-sized** datasets.
- SVM gives **large margin classification**.



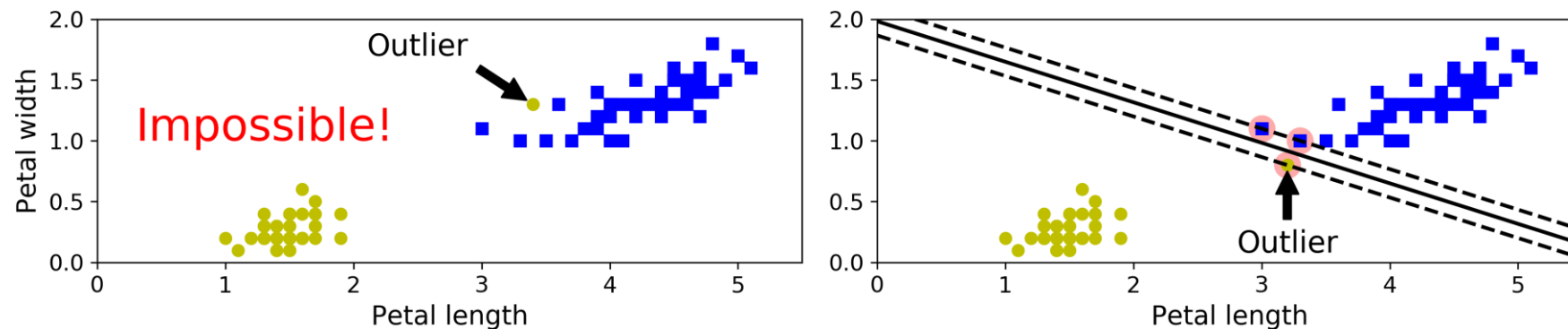
Linear SVM Classification

- The **decision boundary** is fully determined by the instances located on the edge. These instances are called the **support vectors**.
- SVMs are **sensitive** to the **feature scales**.



Soft Margin Classification

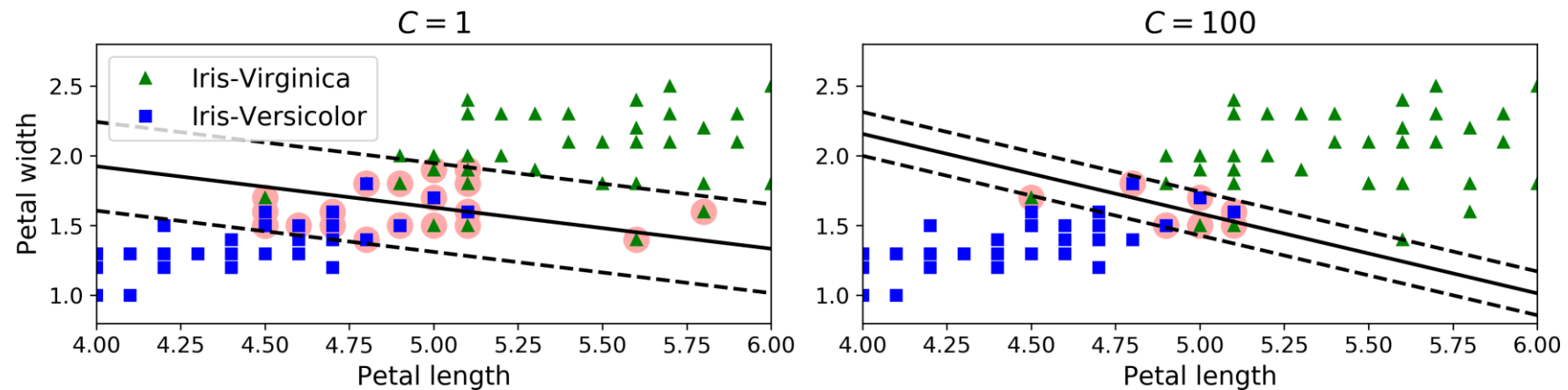
- **Hard margin classification** cannot handle linearly inseparable classes and is sensitive to outliers.



- **Soft margin classification** finds a balance between keeping the margin as large as possible and limiting the margin violations.

Soft Margin Classification

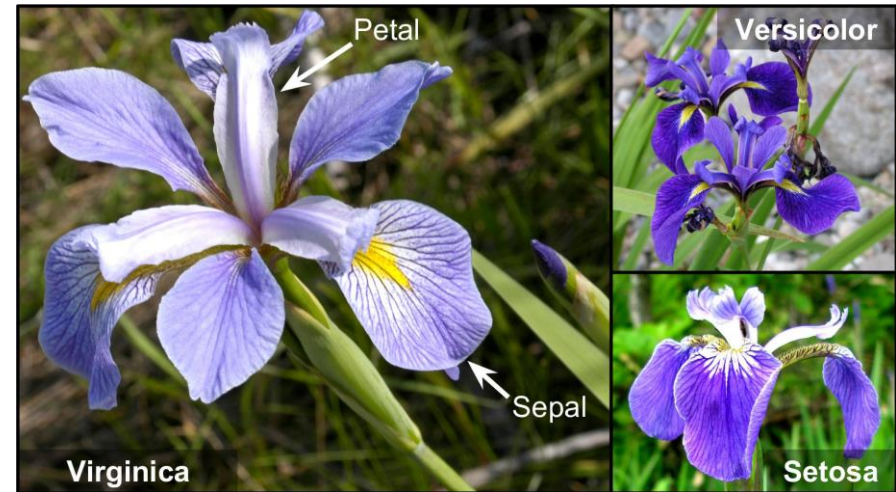
- You can control the number of violations using the **C hyperparameter**.



- If your SVM model is **overfitting**, you can try **regularizing** it by **reducing C**.

Iris Dataset

- A famous dataset that contains the sepal and petal length and width of **150 iris flowers** of three different species: **Setosa**, **Versicolor**, and **Virginica**.




```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
```

SVM Classification Example

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica
svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")), ])
svm_clf.fit(X, y)

>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```



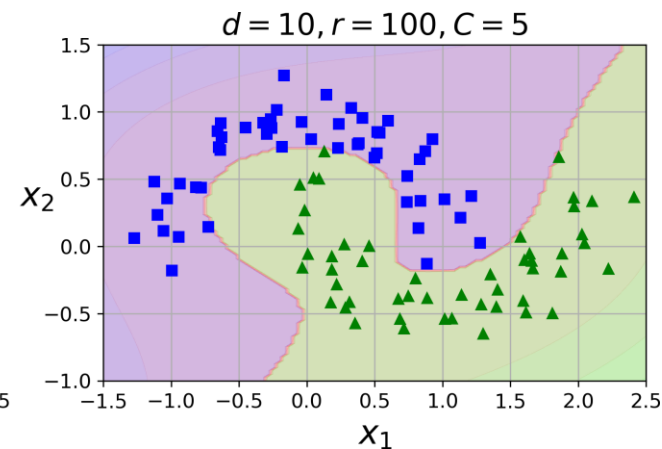
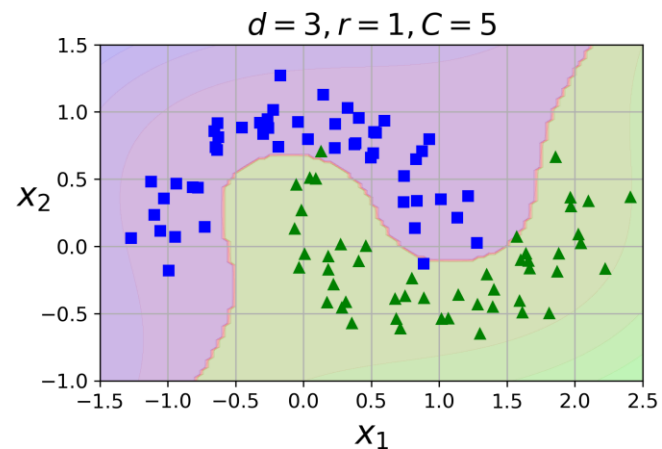
Used for maximum-margin classification.

Nonlinear SVM Classification

- The SVM class supports nonlinear classification using the **kernel** option.

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
])
poly_kernel_svm_clf.fit(X, y)
```

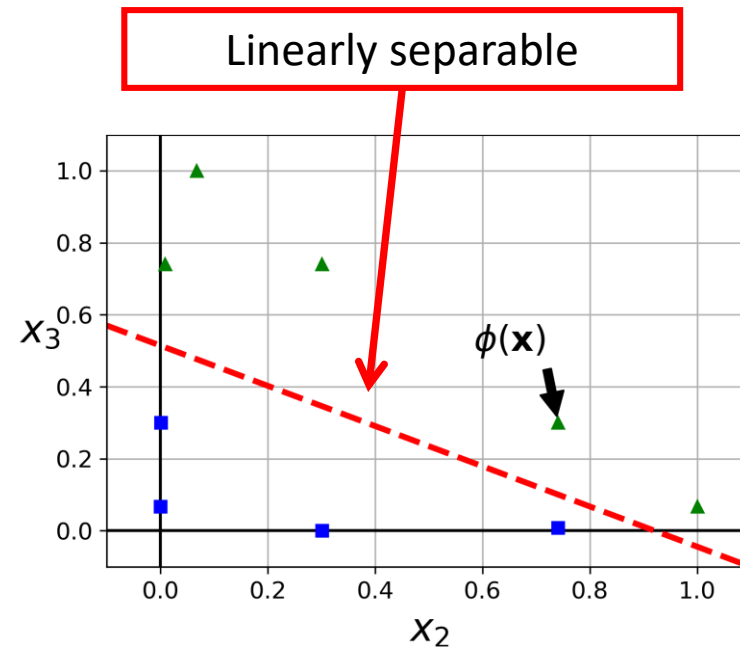
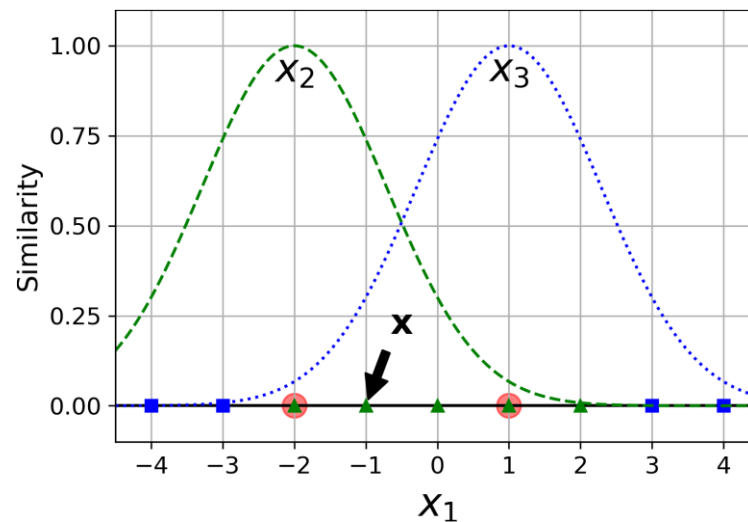
Controls how much the model is influenced by high-degree polynomials versus low-degree



Gaussian Radial Basis Function

$$\phi_\gamma(\mathbf{x}, \ell) = \exp(-\gamma \|\mathbf{x} - \ell\|^2)$$

- The Gaussian RBF can be used to find **similarity features** (x_2 and x_3) of the one-dimensional dataset with two **landmarks** to it at $x_1 = -2$ and $x_1 = 1$



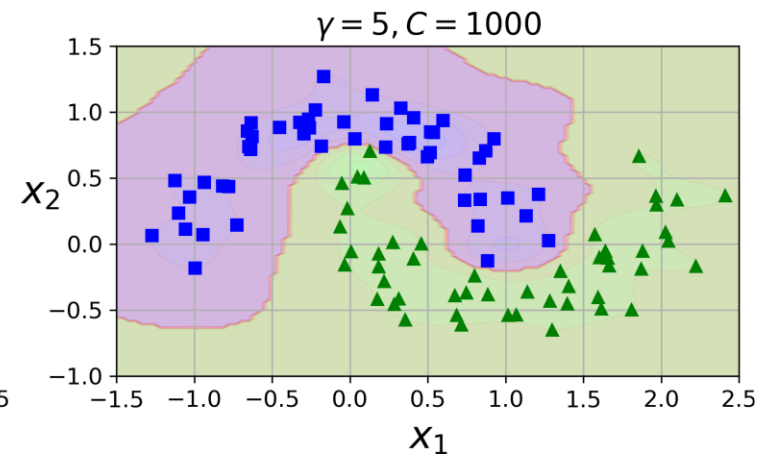
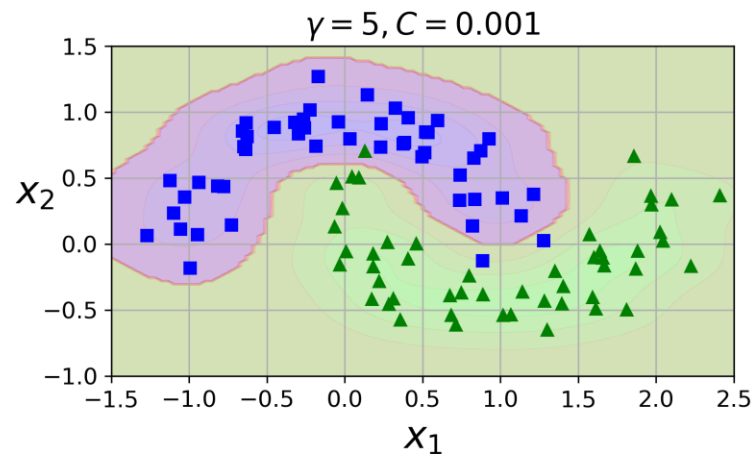
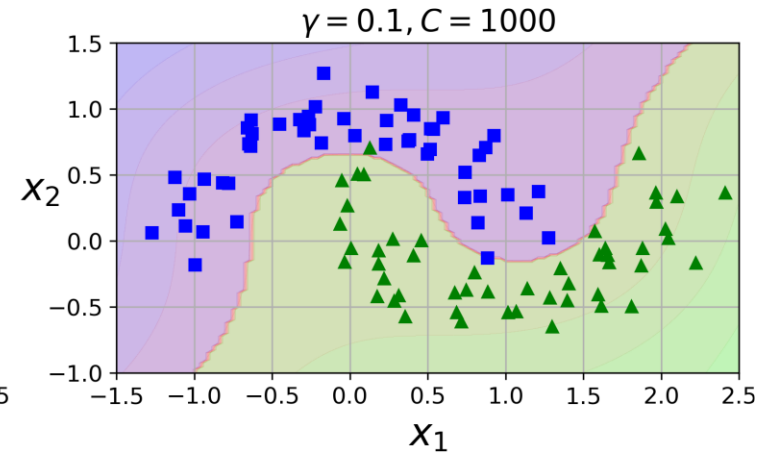
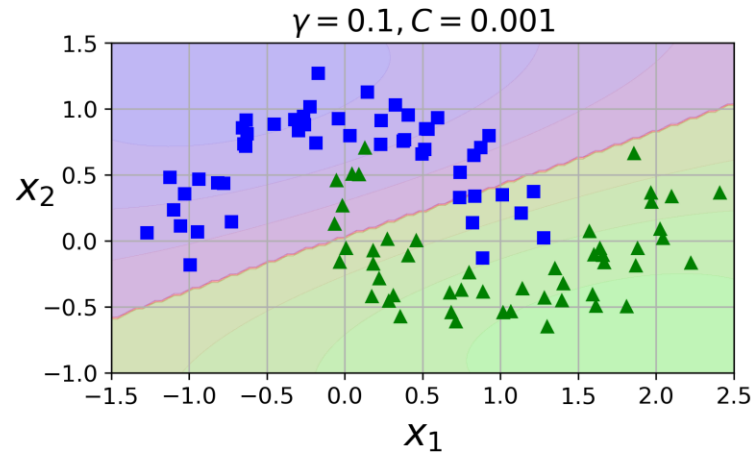
Gaussian RBF Kernel

- Is **popular** with SVM to **solve nonlinear problems**.

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
])
rbf_kernel_svm_clf.fit(X, y)
```

- **Transforms** a training set with m instances and n features to m instances and m features.
- **gamma** and **C** are used for **regularization** with smaller values.

Gaussian RBF Kernel

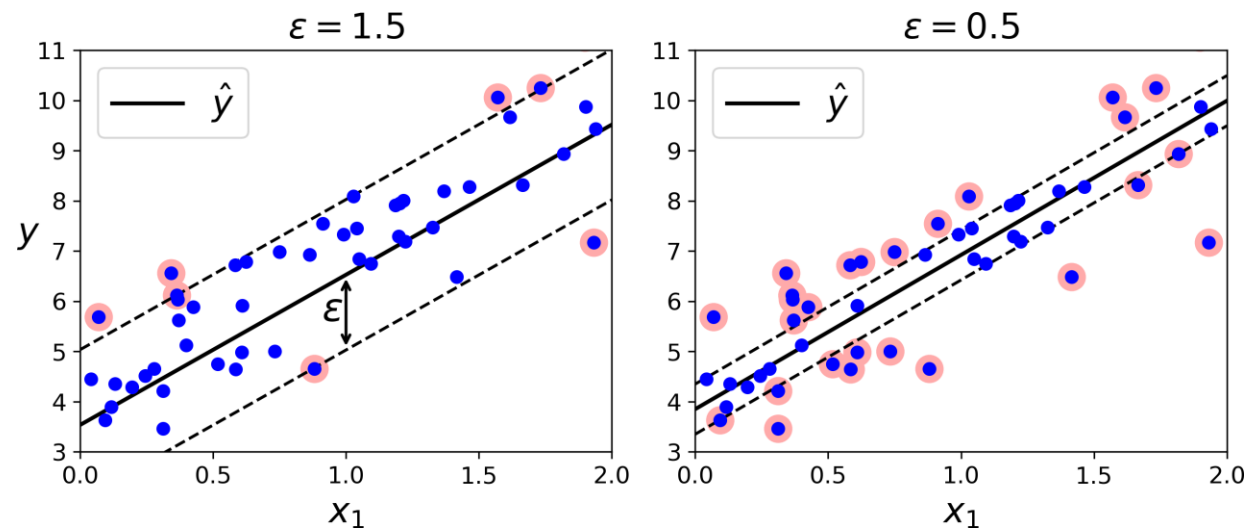


Linear SVM Regression

- Fits as many instances as possible on the margin while limiting margin violations. The width of the street is controlled by a hyperparameter ϵ .

```
from sklearn.svm import LinearSVR
```

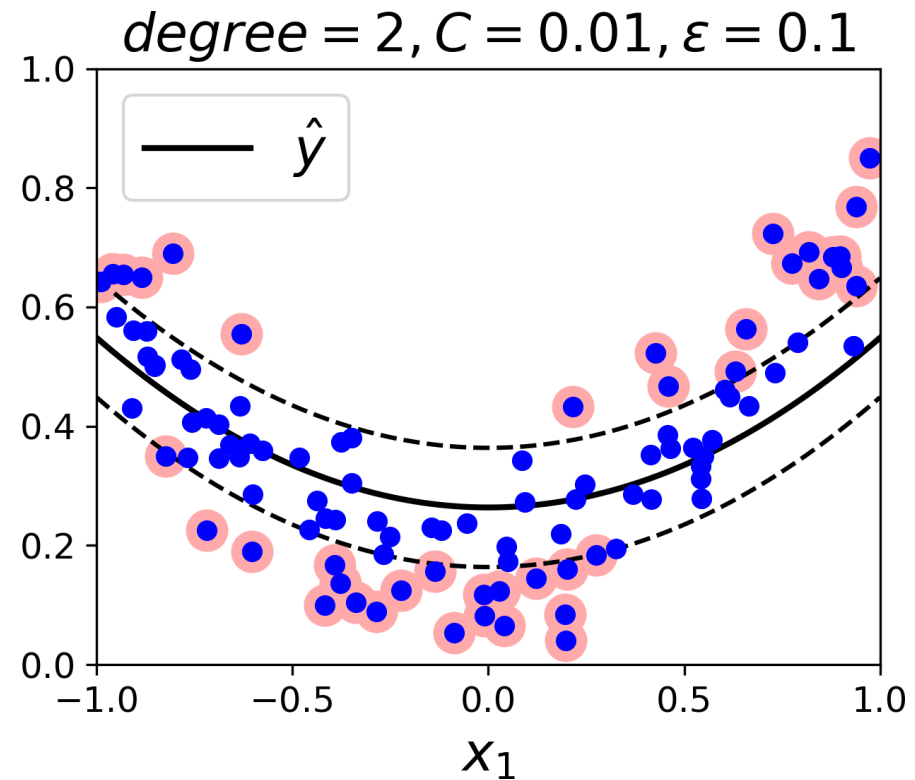
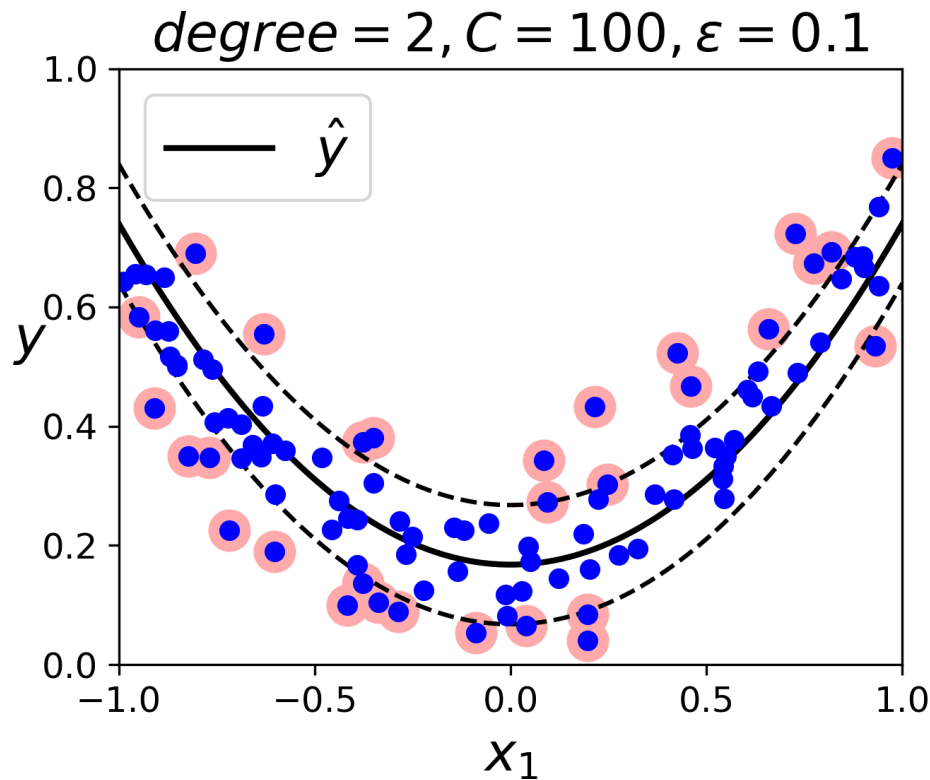
```
svm_reg = LinearSVR(epsilon=1.5)  
svm_reg.fit(X, y)
```



Nonlinear SVM Regression

```
from sklearn.svm import SVR
```

```
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)  
svm_poly_reg.fit(X, y)
```



SVM Conclusion

- The **LinearSVC** has complexity of $O(m \times n)$.
- The **SVC** time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$.
- This algorithm is perfect for complex but small or medium training sets. However, it scales well with the number of features.

Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

Decision Trees

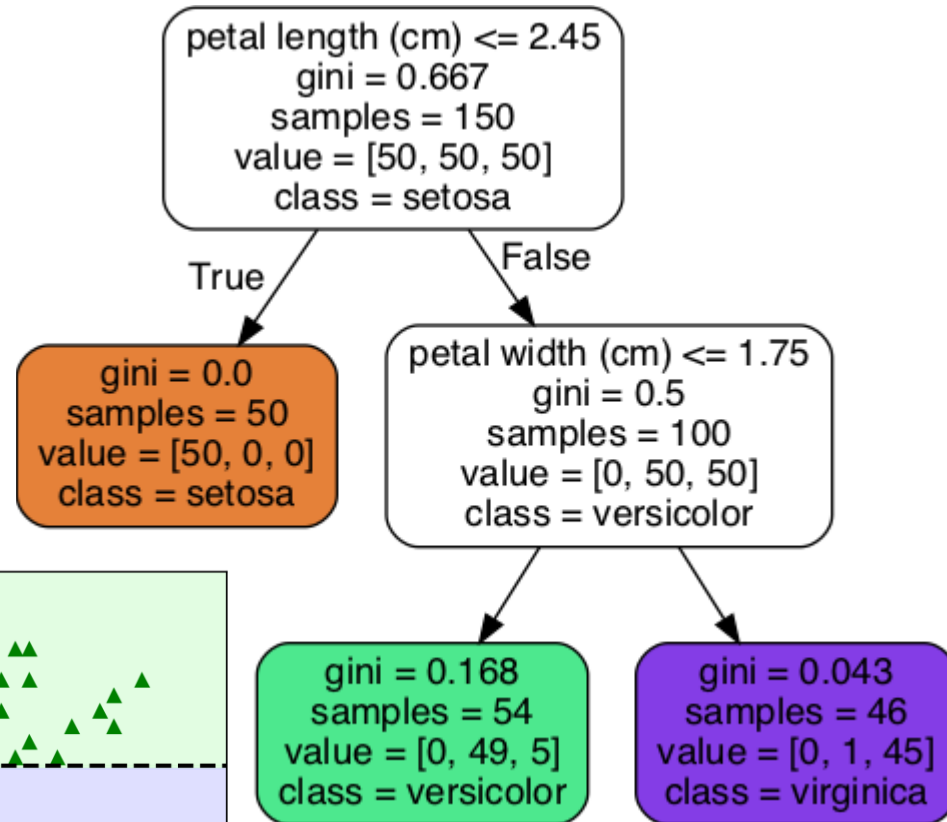
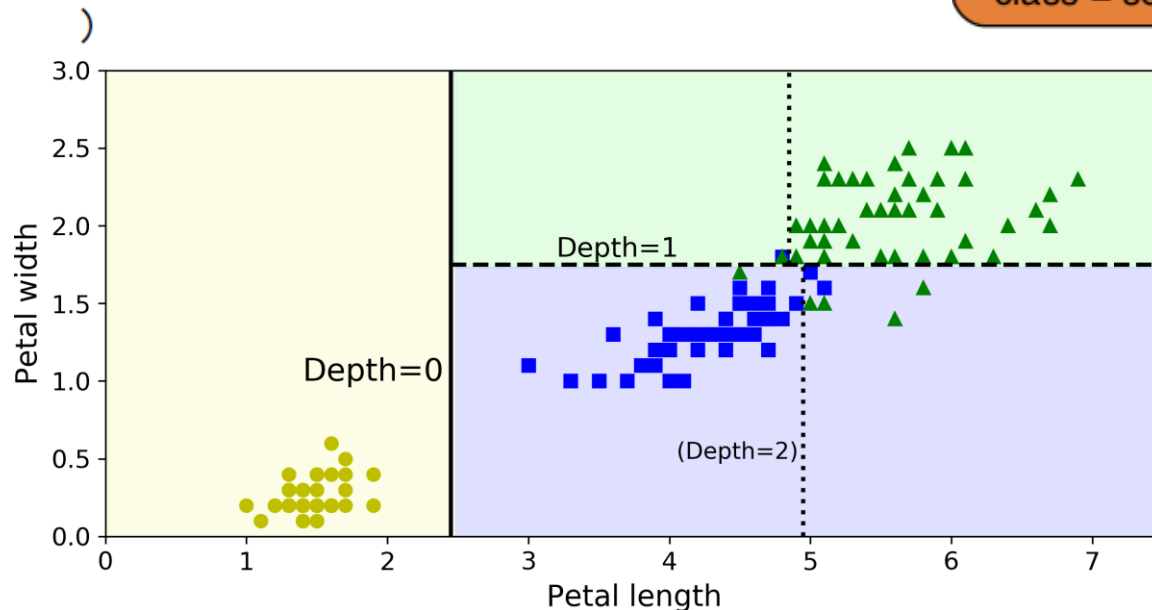
- Decision Trees are **versatile** Machine Learning algorithms that can perform both **classification** and **regression** tasks, and even multioutput tasks.
- They are very powerful algorithms, capable of fitting complex datasets.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Visualizing a Decision Tree

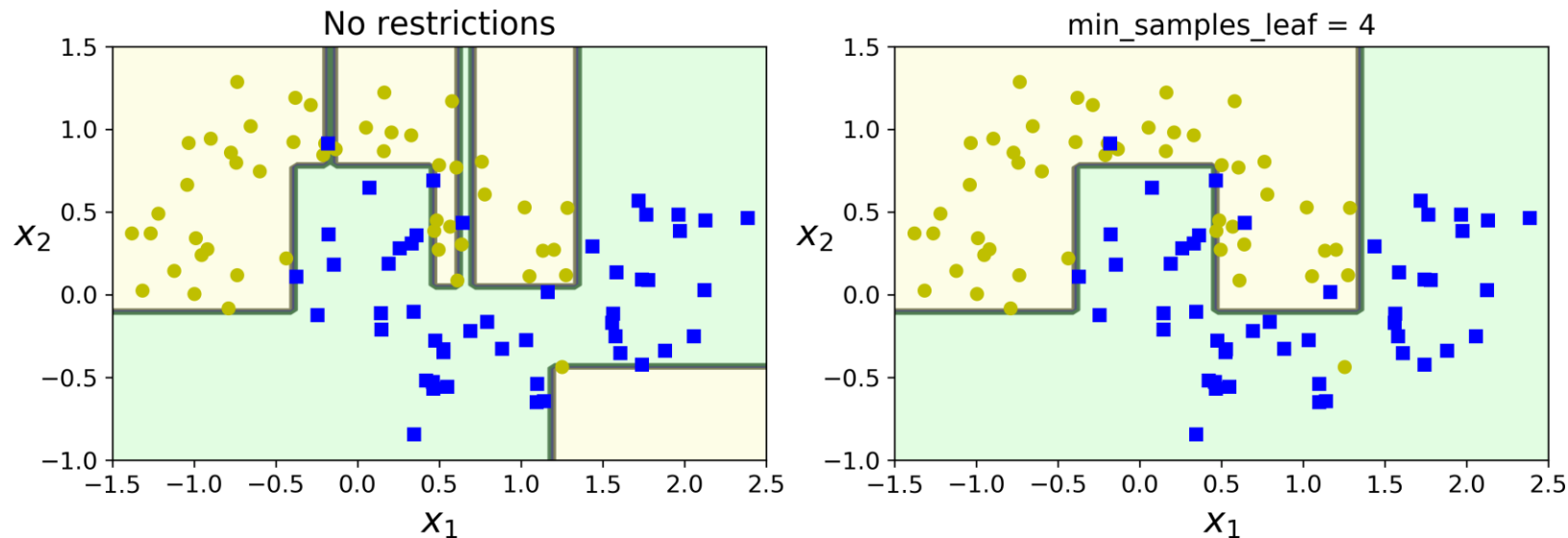
```
from sklearn.tree import export_graphviz
```

```
export_graphviz(  
    tree_clf,  
    out_file=image_path("iris_tree.dot"),  
    feature_names=iris.feature_names[2:],  
    class_names=iris.target_names,  
    rounded=True,  
    filled=True  
)
```



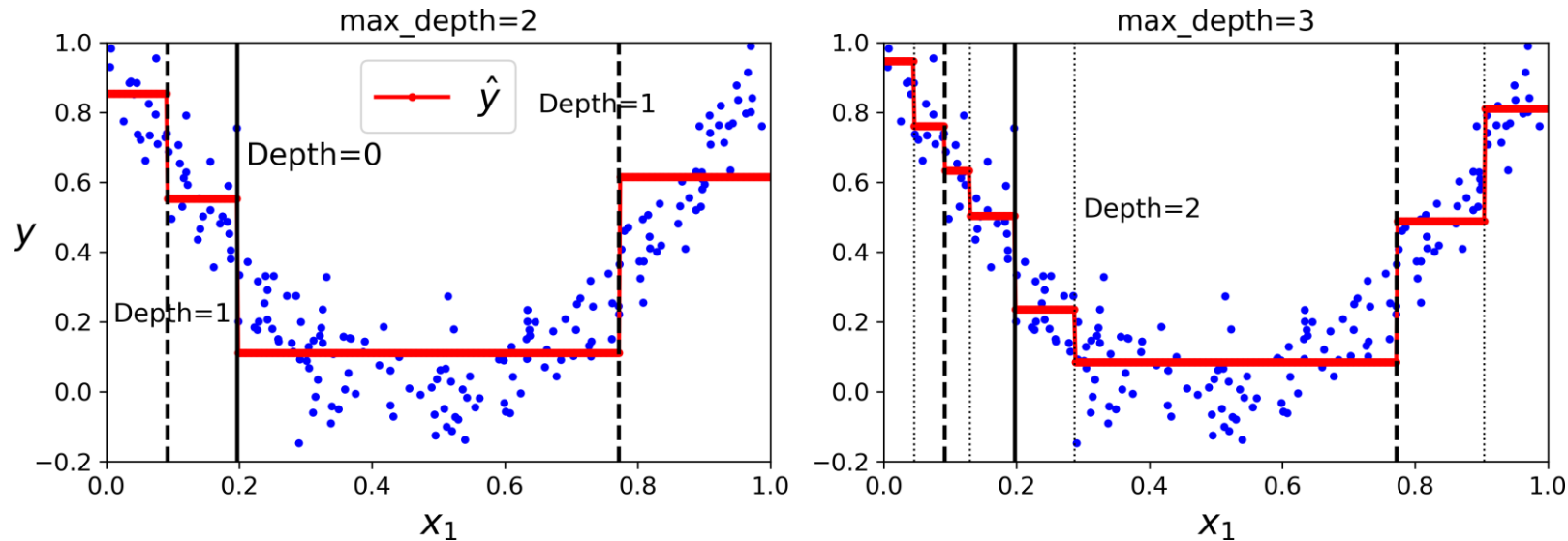
Regularization Hyperparameters

- Increase $\text{min_}*$ or decrease $\text{max_}*$: max_depth=None , $\text{min_samples_split=2}$, $\text{min_samples_leaf=1}$, $\text{min_weight_fraction_leaf=0.0}$, max_features=None , $\text{max_leaf_nodes=None}$



Decision Trees Regression

```
from sklearn.tree import DecisionTreeRegressor  
tree_reg = DecisionTreeRegressor(max_depth=2)  
tree_reg.fit(X, y)
```



Outline

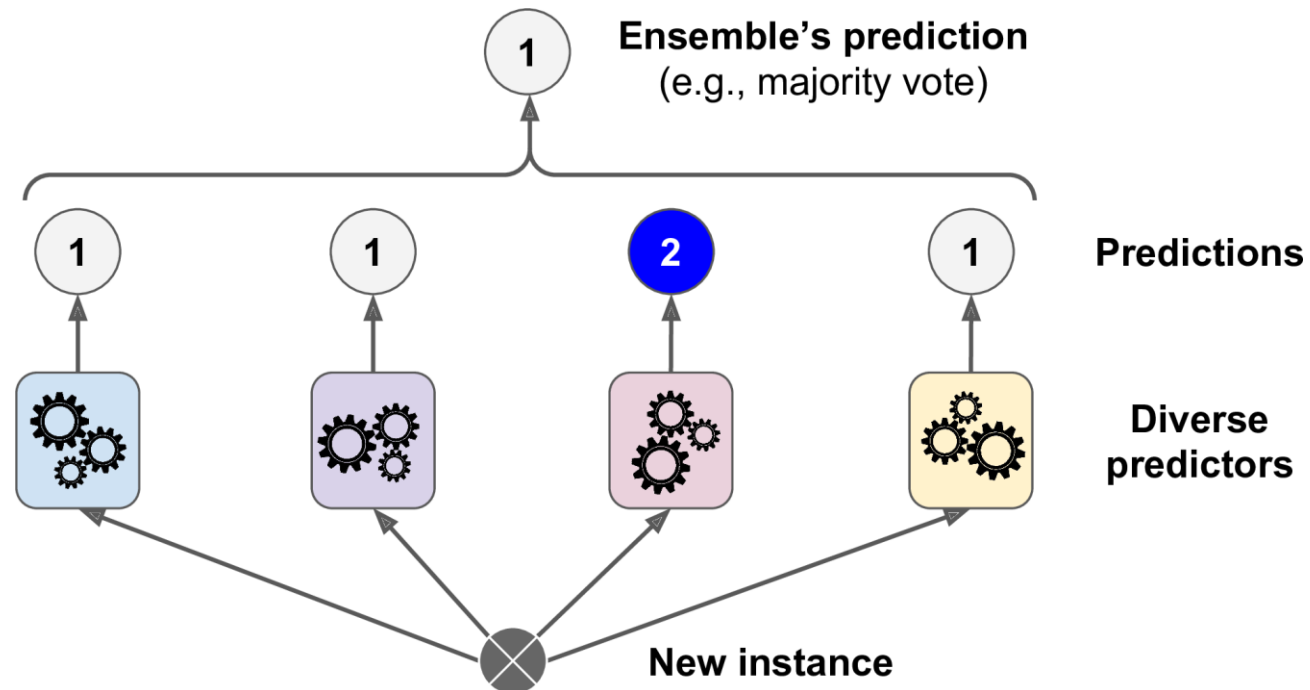
1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

Ensemble Learning and Random Forests

- A group of predictors is called an **ensemble**.
- You can train a group of Decision Tree classifiers, each on a **different random subset** of the training set.
- To make predictions, obtain the predictions of all individual trees, then predict the class that gets the **most votes**.
- Such an ensemble of Decision Trees is called a **Random Forest**.

Voting Classifiers

- If each classifier is a **weak learner** (meaning it does only slightly better than random guessing), the ensemble can be a **strong learner** (achieving high accuracy).



Scikit-Learn Voting Classifier 1/2

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

voting='soft' predict the class with the highest class probability

Scikit-Learn Voting Classifier 2/2

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

Bagging and Pasting

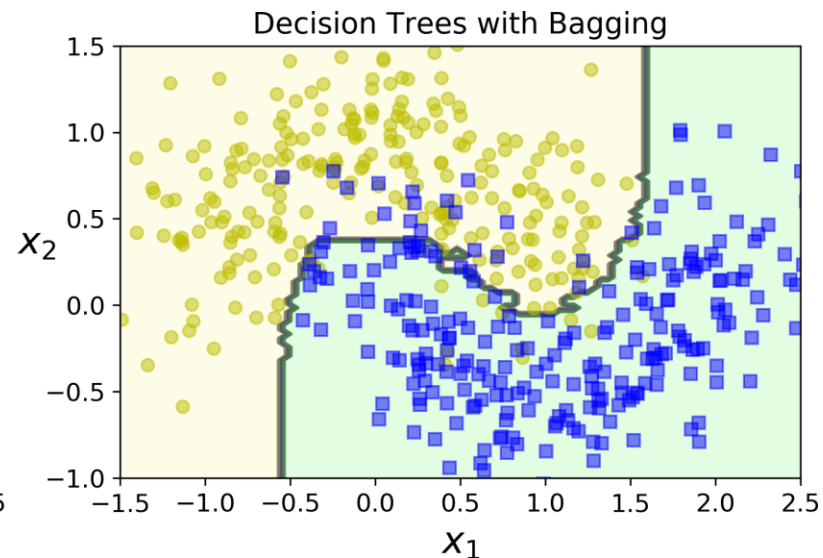
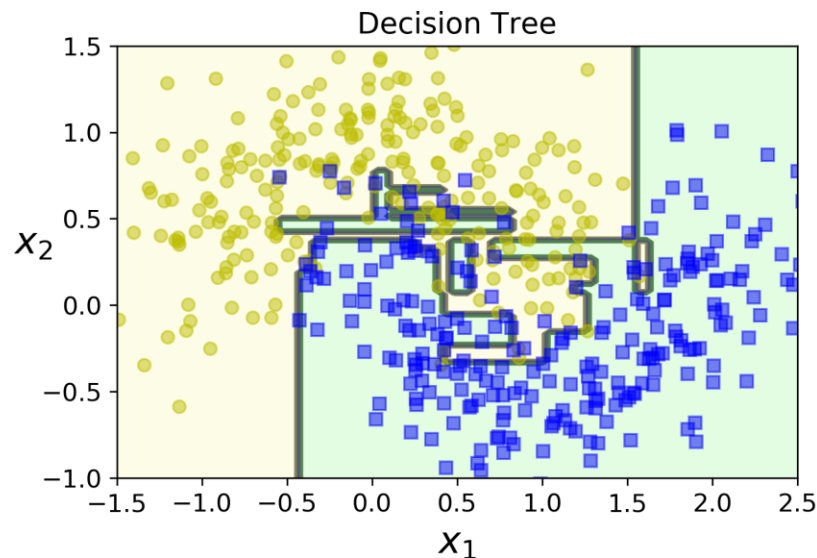
- Use the **same training algorithm** for every predictor, but train them on different random subsets of the training set.
- When sampling is performed **with** replacement, this method is called **bagging** (short for **bootstrap aggregating**).
- When sampling is performed **without** replacement, it is called **pasting**.
- The aggregation function is the most frequent prediction (**hard voting**) for classification, or the **average** for regression.

Bagging and Pasting

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
```

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

with replacement and
use all available cores



Random Forests

- An ensemble of Decision Trees trained via the bagging with **max_samples** set to the size of the training set, and choosing the best random splits.

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

- Equivalent to:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

Outline

1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

Exercises

1. Train an **SVM classifier** on the **MNIST** dataset. Since SVM classifiers are binary classifiers, you will need to use one-versus-all to classify all 10 digits. You may want to tune the hyperparameters using small validation sets to speed up the process. What accuracy can you reach?

Exercises

2. Train and fine-tune a **Decision Tree** for the **moons dataset**.
 - a) Generate a moons dataset using `make_moons(n_samples=10000, noise=0.4)`.
 - b) Split it into a training set and a test set using `train_test_split()`.
 - c) Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.
 - d) Train it on the full training set using these hyperparameters, and measure your model's performance on the test set. You should get roughly 85% to 87% accuracy.

Exercises

3. Load the **MNIST** data and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for validation, and 10,000 for testing). Then train various classifiers, such as a **Random Forest classifier**, an **Extra-Trees** classifier, and an **SVM**. Next, try to combine them into an **ensemble** that outperforms them all on the validation set, using a **soft** or **hard** voting classifier. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?

Summary

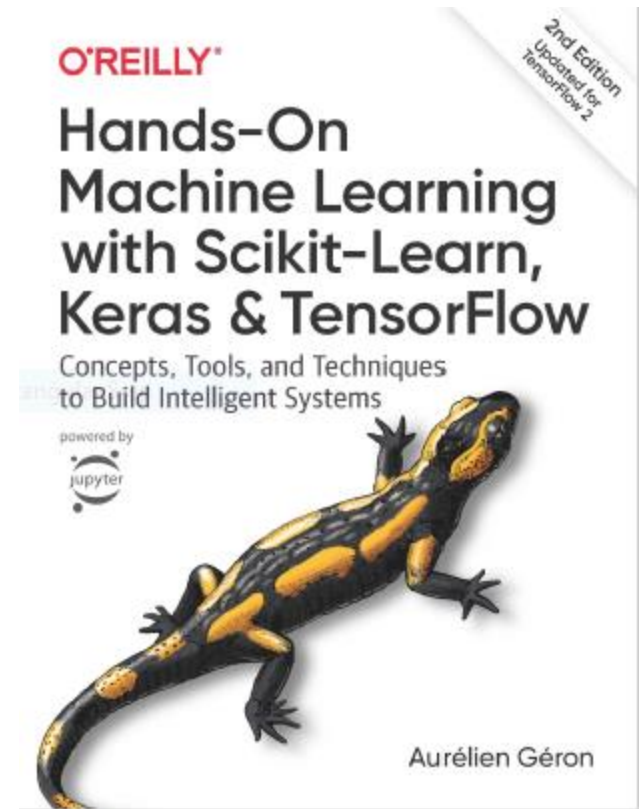
1. k-Nearest Neighbors
2. Support Vector Machines
3. Decision Trees
4. Ensemble Learning and Random Forests
5. Exercises

Unsupervised Learning and Clustering

Prof. Gheith Abandah

Reference

- Chapter 8: **Dimensionality Reduction**
 - Chapter 9: **Unsupervised Learning Techniques**
-
- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>



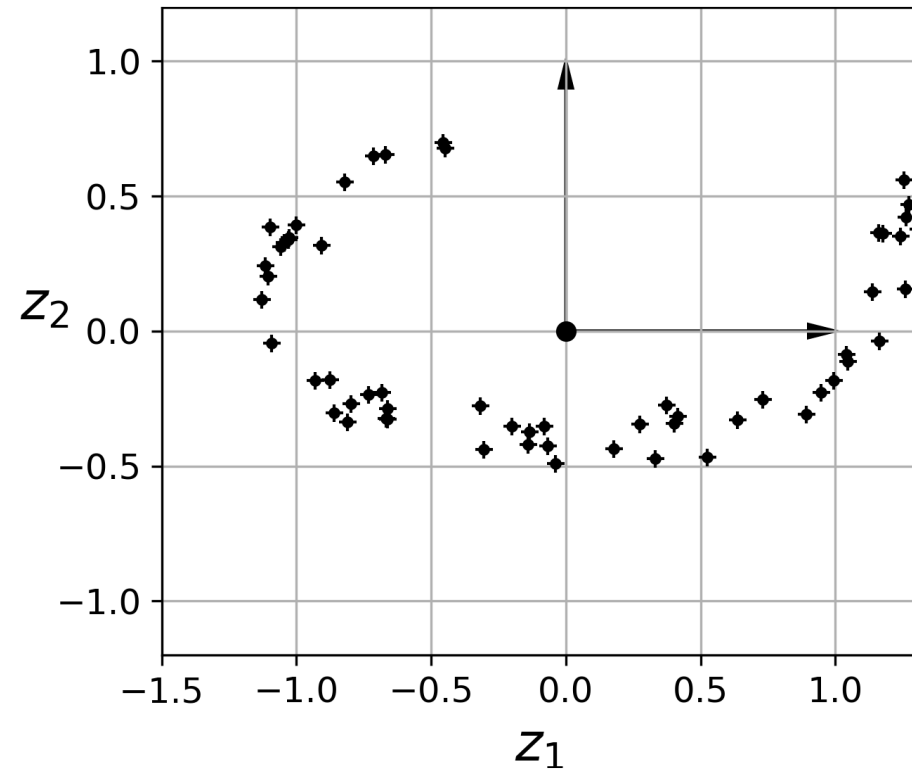
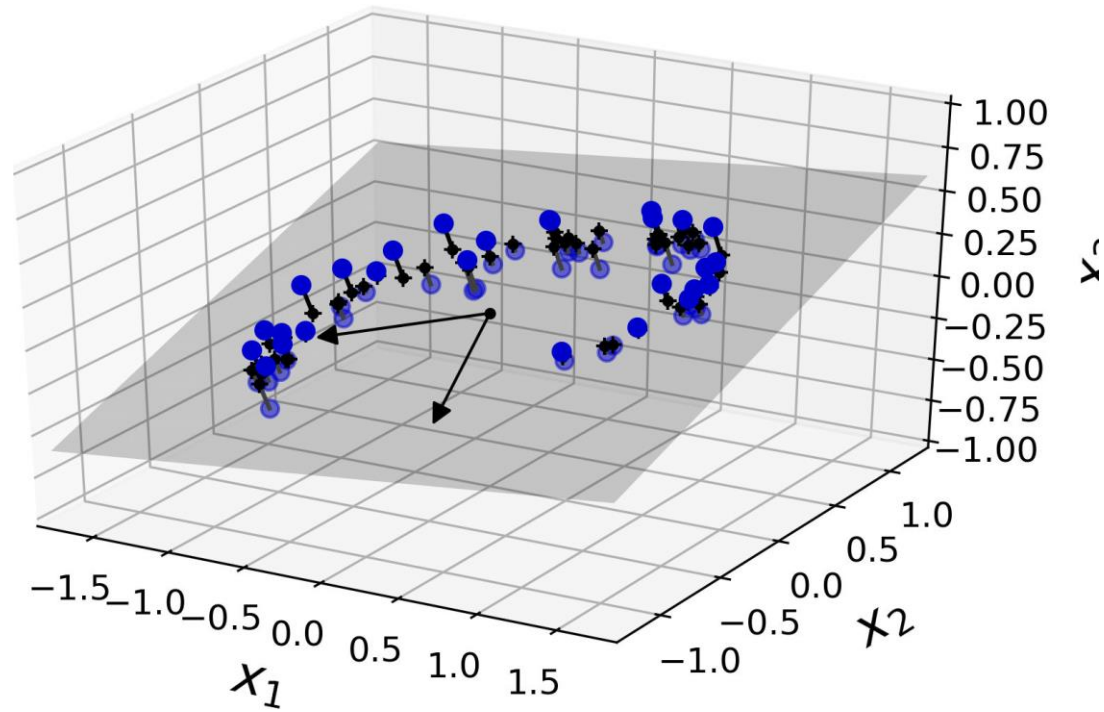
Outline

- Dimensionality Reduction
 - Projection and Manifold
 - Principal Component Analysis (PCA)
- Unsupervised Learning
- Clustering
 - K-Means
 - DBSCAN
- Gaussian Mixtures and Anomaly Detection
- Exercises

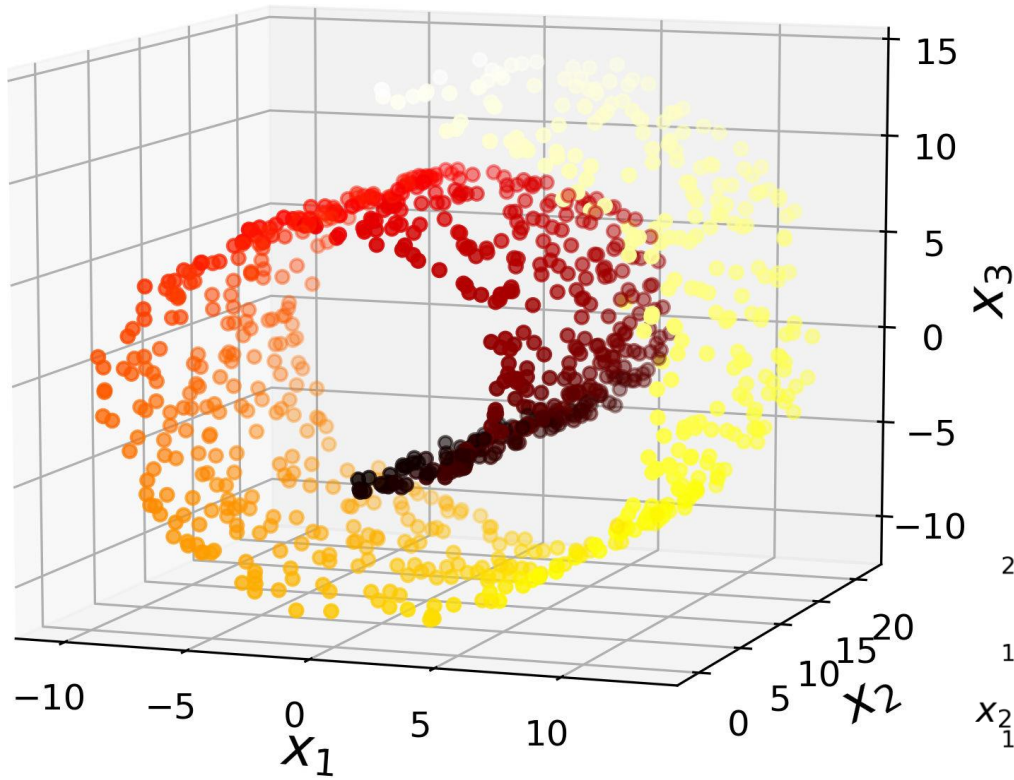
Dimensionality Reduction

- Many Machine Learning problems involve **thousands** or even **millions of features** for each training instance.
- All these features make training extremely **slow** and make it much **harder** to find a good solution.
- This problem is often referred to as the **curse of dimensionality**.
- **Dimensionality reduction approaches**
 - **Drop** not useful features
 - **Merge** correlated features
 - **Projection** and manifold
 - **Transform** features

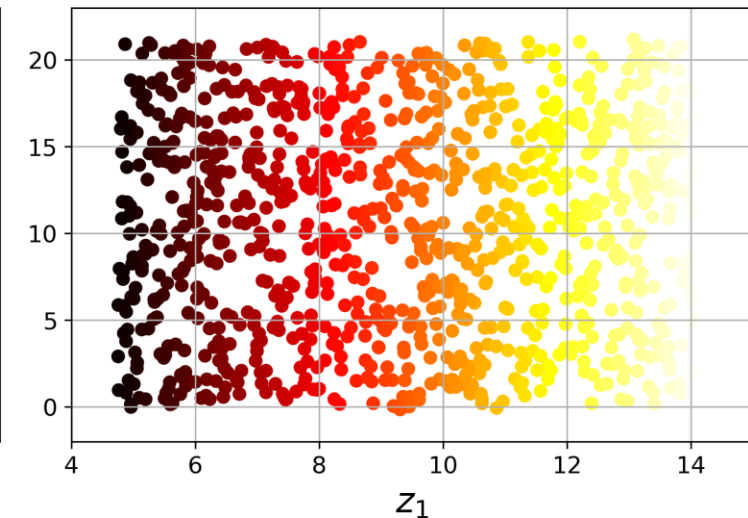
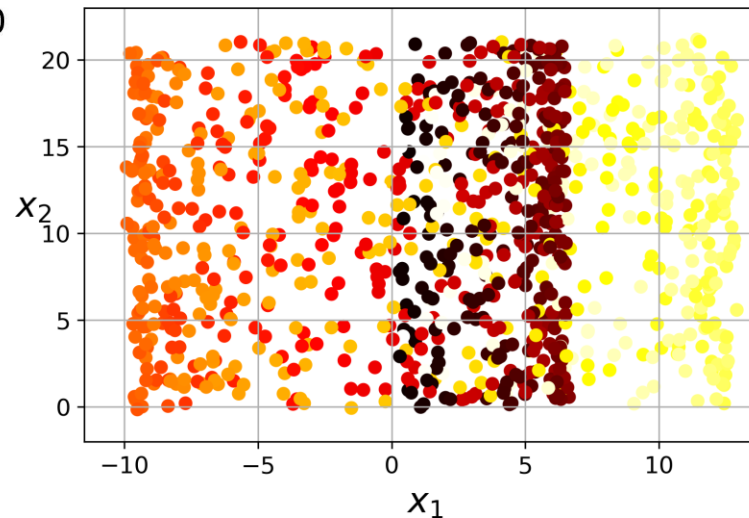
Projection and Manifold



Projection and Manifold

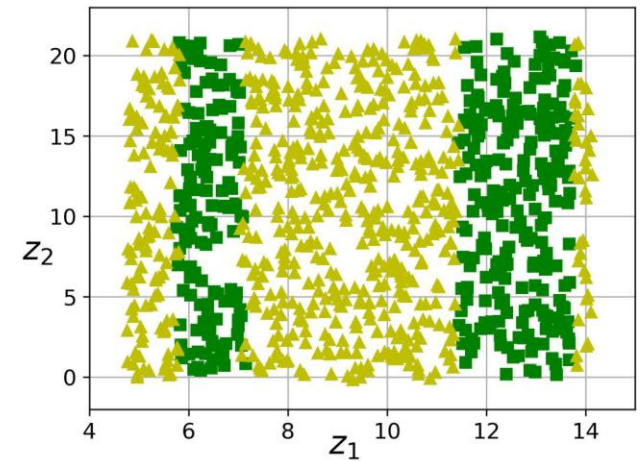
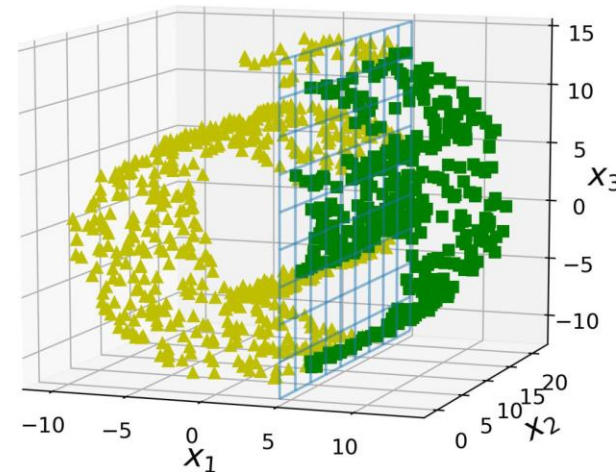
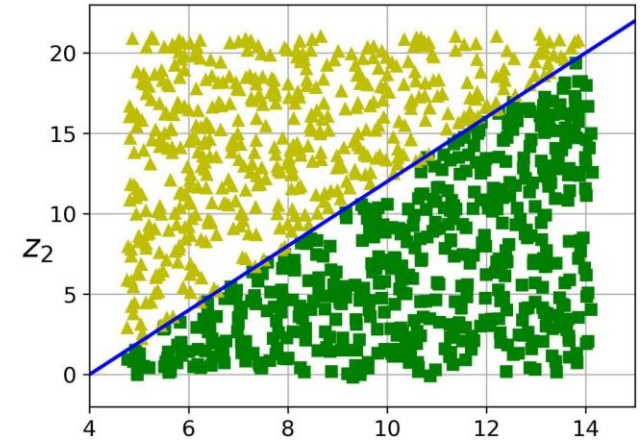
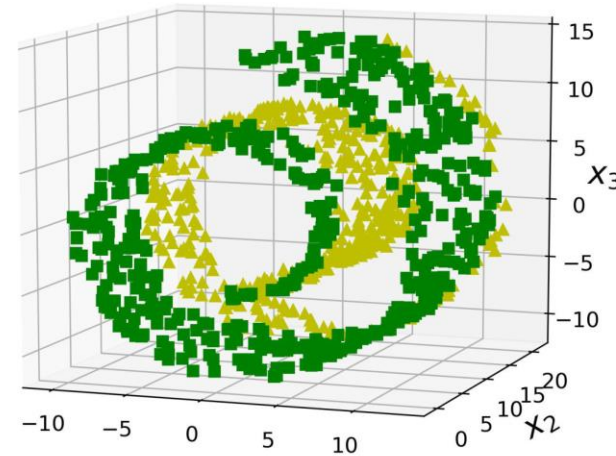


- Simply projecting onto a plane may not give better solution.
- Projecting to a proper manifold is better.



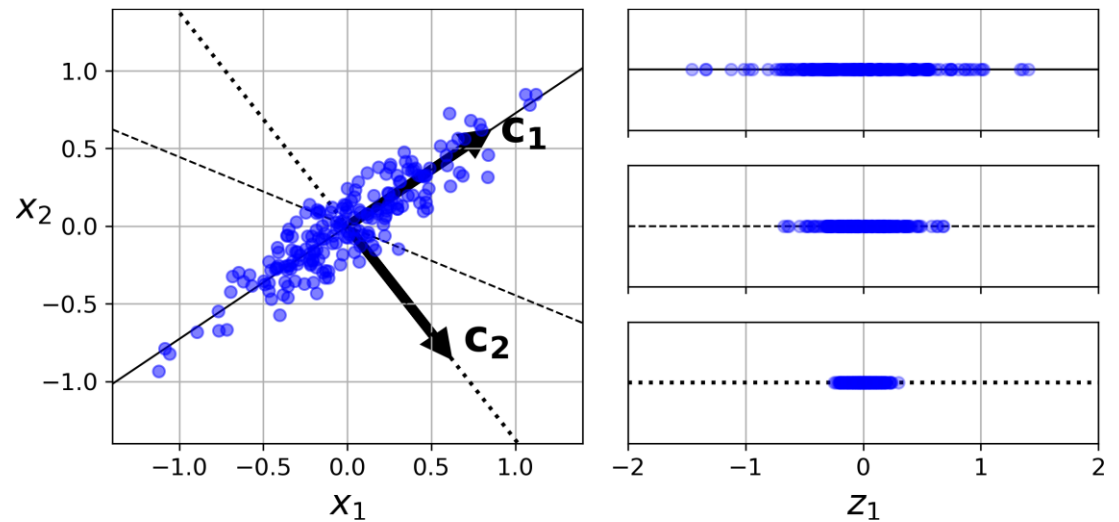
Projection and Manifold

- The decision boundary may not always be simpler with lower dimensions.



Principal Component Analysis (PCA)

- Is the most **popular** dimensionality reduction algorithm.
- First it identifies the **hyperplane** that lies **closest to the data**, and then it **projects** the data onto it.
- PCA identifies the **axis** that accounts for the **largest amount of variance** in the training set. Then it finds the **next orthogonal axes** that accounts for the largest amount of remaining variance.



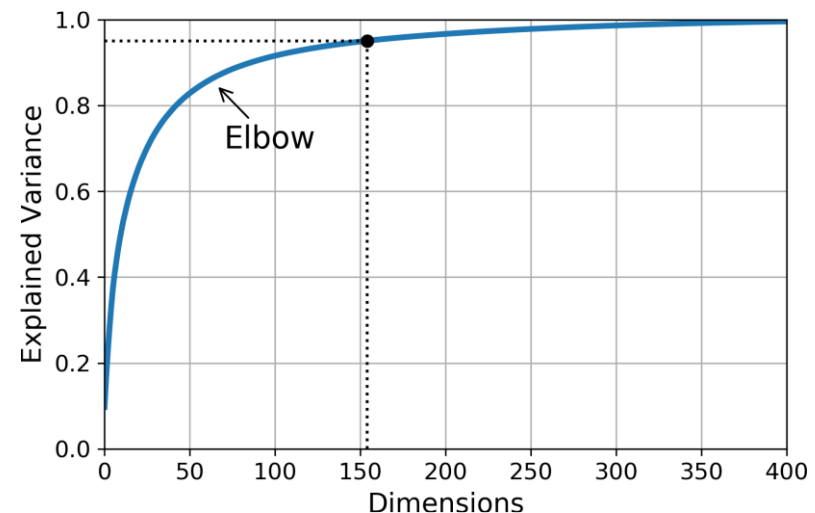
Principal Component Analysis (PCA)

- Use PCA to reduce the dimensionality of the dataset down to **two dimensions**.
- Instead of specifying the number of principal components you want to preserve, you can set **n_components** to be a float between **0.0** and **1.0**, indicating the ratio of variance you wish to preserve.

```
from sklearn.decomposition import PCA  
pca = PCA(n_components = 2)  
X2D = pca.fit_transform(X)
```

3-D

```
pca = PCA(n_components=0.95)  
X_reduced = pca.fit_transform(X_train)
```



Outline

- Dimensionality Reduction
 - Projection and Manifold
 - Principal Component Analysis (PCA)
- **Unsupervised Learning**
- Clustering
 - K-Means
 - DBSCAN
- Gaussian Mixtures and Anomaly Detection
- Exercises

Unsupervised Learning

If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake.

Yann LeCun

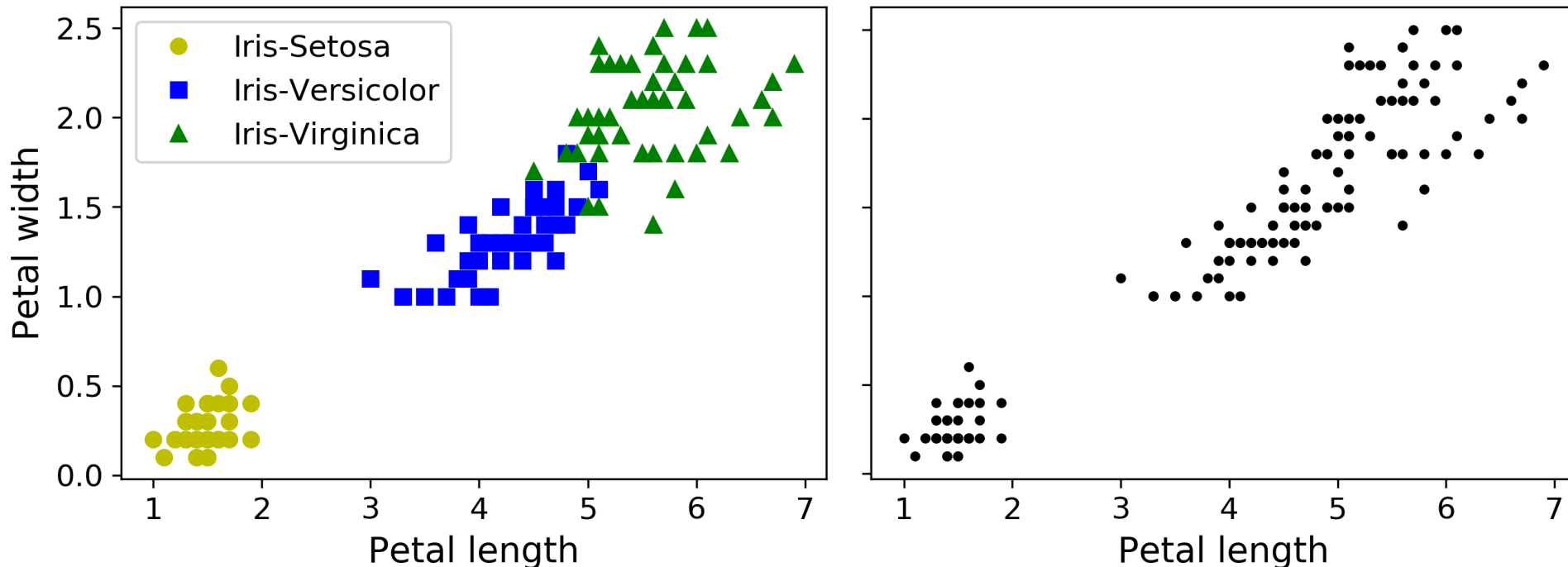
- **Example:** System that takes a few pictures of each item on a manufacturing production line and detects which items are defective.

Outline

- Dimensionality Reduction
 - Projection and Manifold
 - Principal Component Analysis (PCA)
- Unsupervised Learning
- Clustering
 - K-Means
 - DBSCAN
- Gaussian Mixtures and Anomaly Detection
- Exercises

Clustering

- The task of **identifying similar instances** and assigning them to clusters, i.e., groups of similar instances.
- **Classification** (left) versus **clustering** (right)

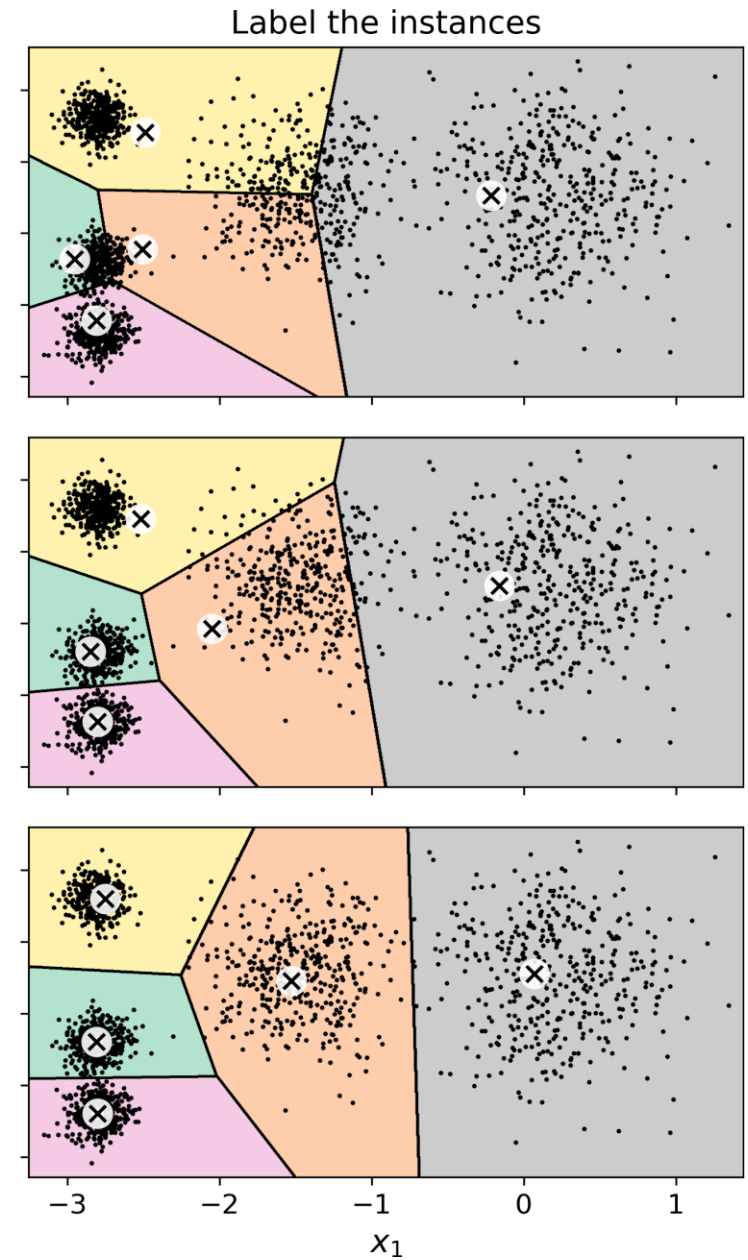
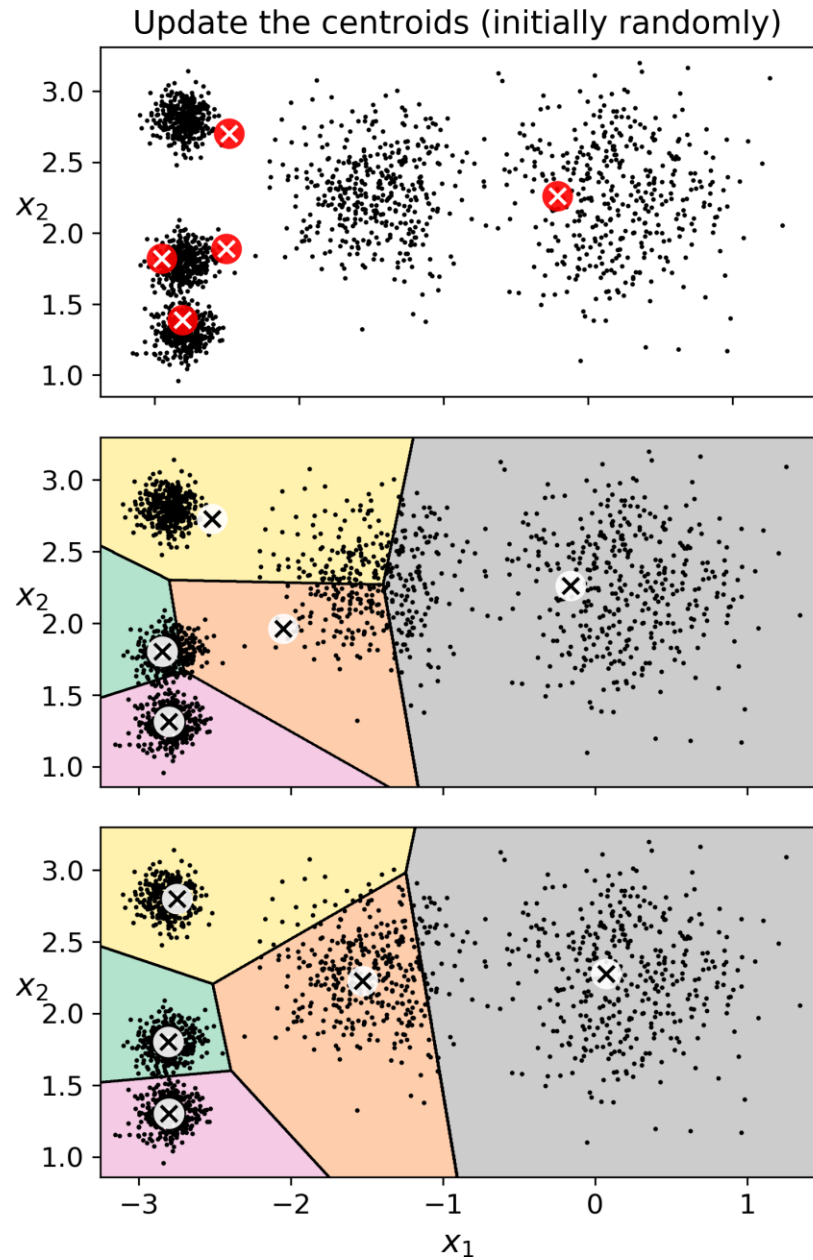


Clustering Applications

- **Customer segmentation**: useful for recommender systems.
- **Data analysis**: discover clusters of similar instances as it is often easier to analyze clusters separately.
- **Dimensionality reduction**: find affinity features to the found clusters
- **Anomaly detection**: any instance that has a low affinity to all the clusters is likely to be an anomaly.
- **Semi-supervised learning**: perform clustering and propagate the labels to all the instances in the same cluster.
- **Search engines** for images
- **Image segmentation**

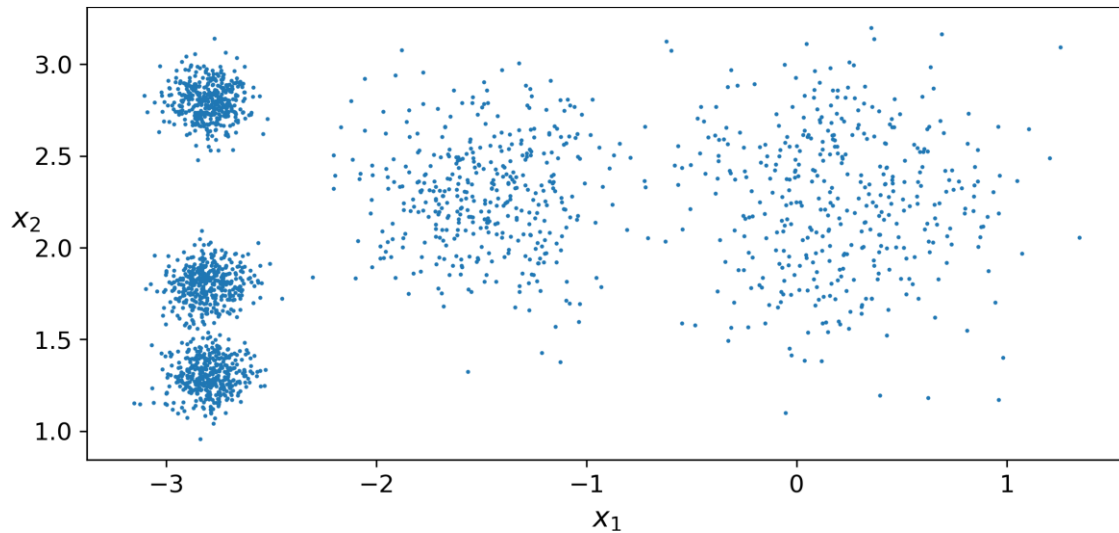
K-Means

- **Quick** and **efficient** algorithm
- **Scale** before clustering
- Need to **specify** the **number of clusters**



K-Means

- Cluster to 5 clusters



```
from sklearn.cluster import KMeans
```

```
k = 5
```

```
kmeans = KMeans(n_clusters=k)
```

```
y_pred = kmeans.fit_predict(X)
```

```
y_pred
```

```
array([4, 0, 1, ..., 2, 1, 0],  
      dtype=int32)
```

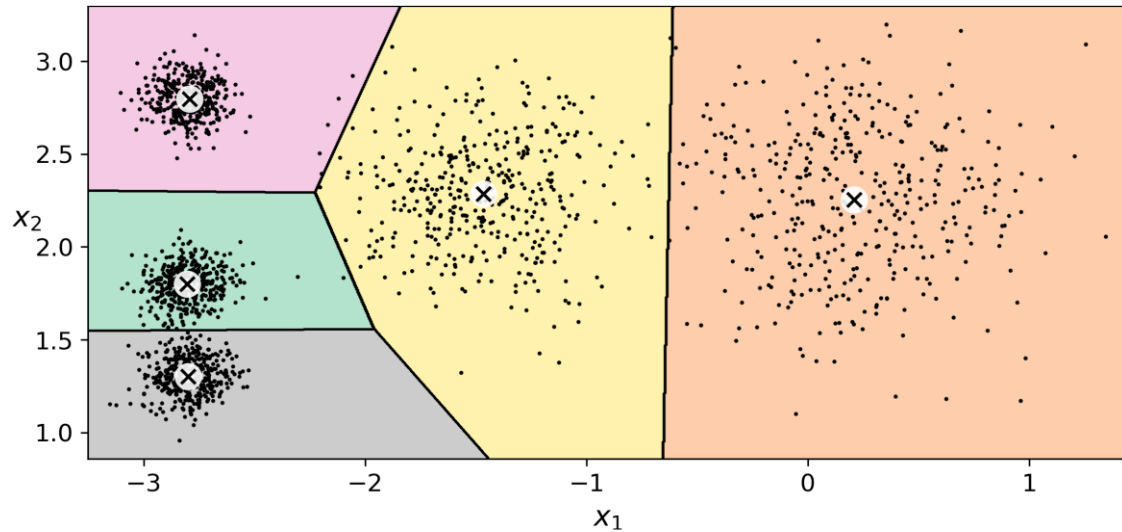
```
# Hard clustering:
```

```
X_new = np.array([[0, 2], [-3, 3]])
```

```
kmeans.predict(X_new)
```

```
array([1, 2], dtype=int32)
```

K-Means



```
kmeans.cluster_centers_  
array([[ -2.80389616,  1.80117999],  
       [  0.20876306,  2.25551336],  
       [-2.79290307,  2.79641063],  
       [-1.46679593,  2.28585348],  
       [-2.80037642,  1.30082566]])
```

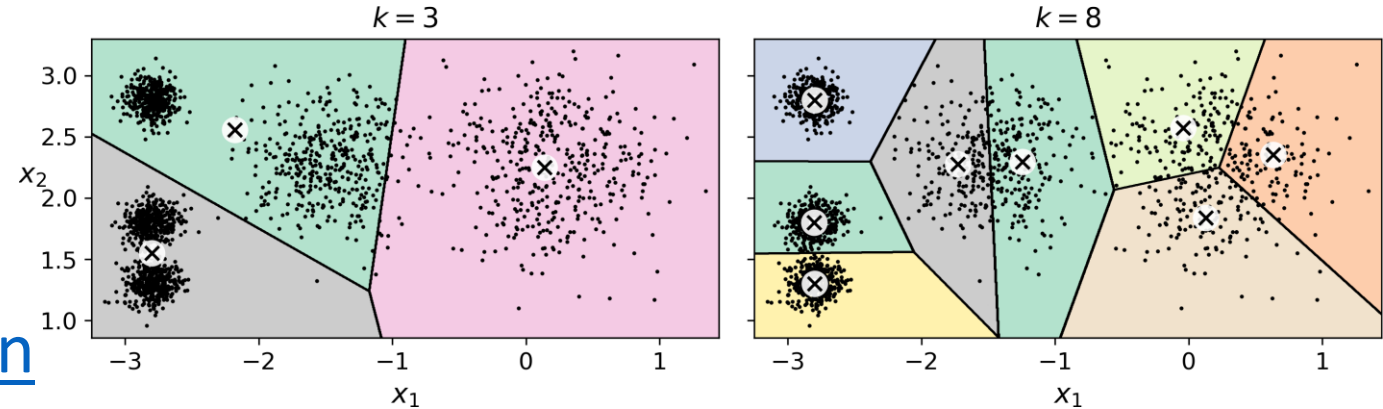
Can be a dimensionality reduction technique.

Soft clustering, a score per # cluster:

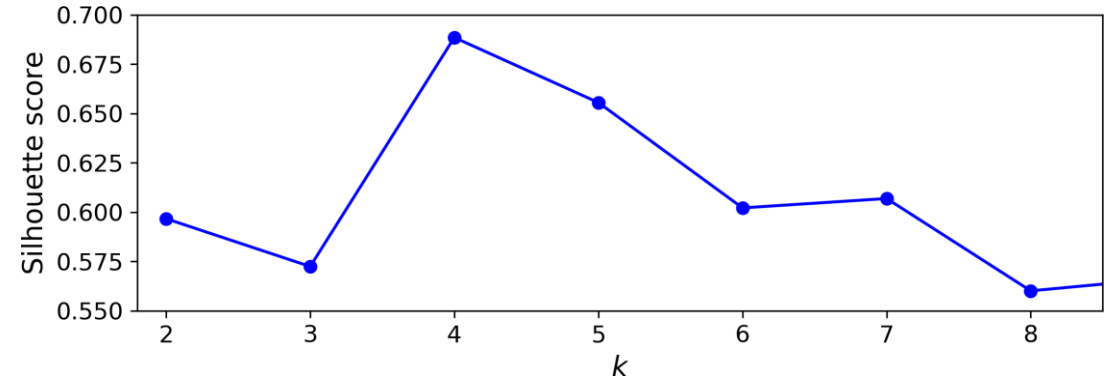
```
kmeans.transform(X_new)  
array([[2.81093633, 0.32995317,  
       2.9042344 , 1.49439034,  
       2.88633901],  
       [1.21475352, 3.29399768,  
       0.29040966, 1.69136631,  
       1.71086031])
```


K-Means

- It is important to specify the **right** number of clusters k .
- Find k that gives highest mean silhouette coefficient.



```
from sklearn.metrics import  
silhouette_score  
silhouette_score(X, kmeans.labels_)  
0.655517642572828
```



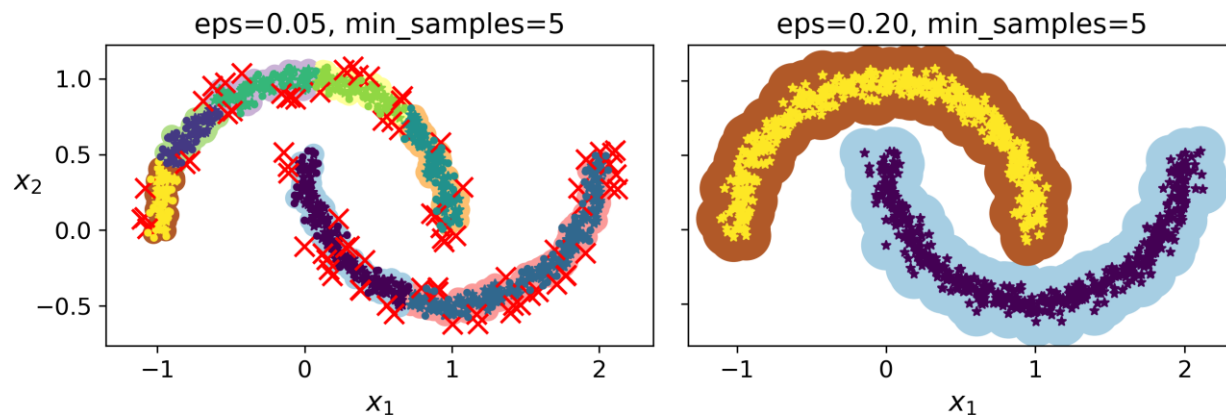
DBSCAN

- Defines **clusters as continuous regions** of high density.
 - Works well if all the **clusters are dense enough**, and they are well **separated by low-density regions**.
 - Behaves well when the clusters have **varying sizes** or **non-spherical** shapes.
- **Algorithm**
 - For each instance, counts how many instances are located within a small distance **ϵ -neighborhood**.
 - If an instance has at least **min_samples** instances in its ϵ -neighborhood, then it is considered a **core instance**.
 - All instances in the neighborhood of a core instance belong to the same cluster. This may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
 - Any instance that is not a core instance and does not have one in its neighborhood is considered an **anomaly (-1)**.

Can detect anomalies

DBSCAN

- Cluster the **moons** dataset

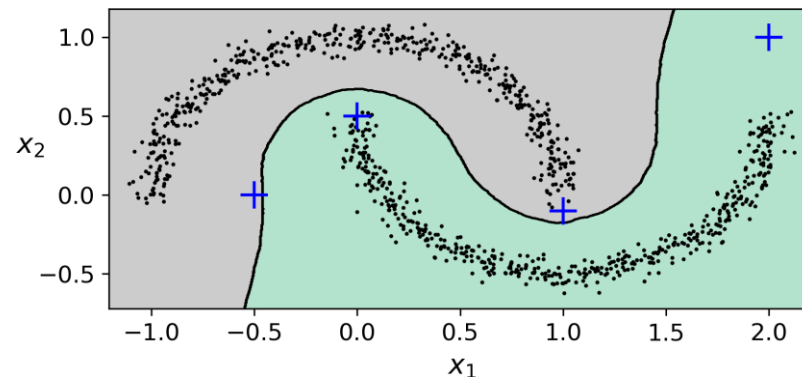


```
from sklearn.cluster import DBSCAN
from sklearn.datasets import
    make_moons
X, y = make_moons(n_samples=1000,
                  noise=0.05)
dbscan = DBSCAN(eps=0.2,
                min_samples=5)
dbscan.fit(X)
```

DBSCAN

- DBSCAN class does not have a **predict()** method.
- Can use other classifiers.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
X_new = np.array([[ -0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
array([1, 0, 1, 0])
```



Outline

- Dimensionality Reduction
 - Projection and Manifold
 - Principal Component Analysis (PCA)
- Unsupervised Learning
- Clustering
 - K-Means
 - DBSCAN
- **Gaussian Mixtures and Anomaly Detection**
- Exercises

Gaussian Mixtures

- A **Gaussian mixture model (GMM)** is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown.
- Scikit-Learn's **GaussianMixture** class, given the dataset **X**, can estimate the weights **φ** and all the distribution parameters **$\mu^{(1)}$** to **$\mu^{(k)}$** and **$\Sigma^{(1)}$** to **$\Sigma^{(k)}$** .

```
from sklearn.mixture import GaussianMixture
gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

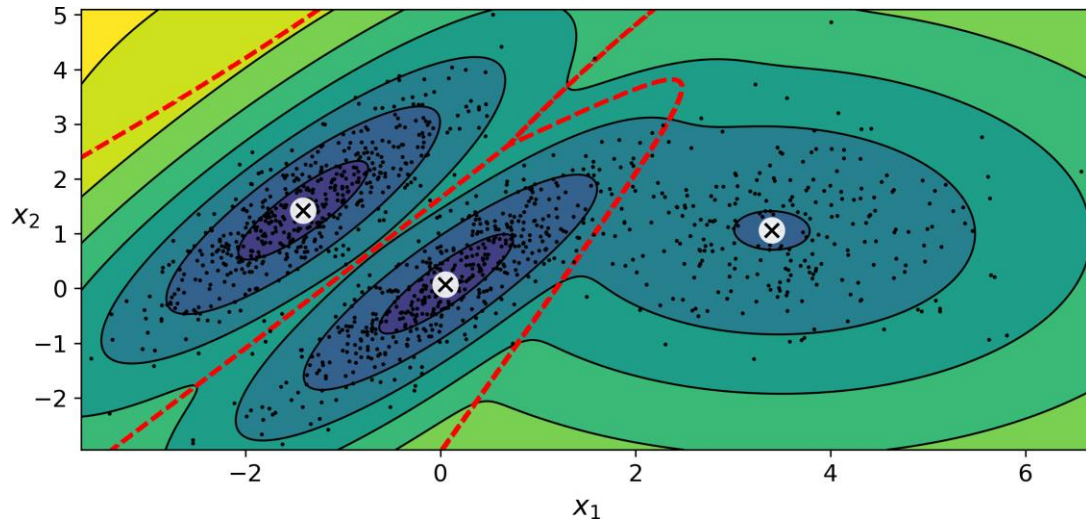
Gaussian Mixtures

```
gm.converged_
```

```
True
```

```
gm.n_iter_
```

```
3
```



```
gm.weights_
```

```
array([0.20965228, 0.4000662,  
       0.39028152])
```

```
gm.means_
```

```
array([[ 3.39909717,  1.05933727],  
       [-1.40763984,  1.42710194],  
       [ 0.05135313,  0.07524095]])
```

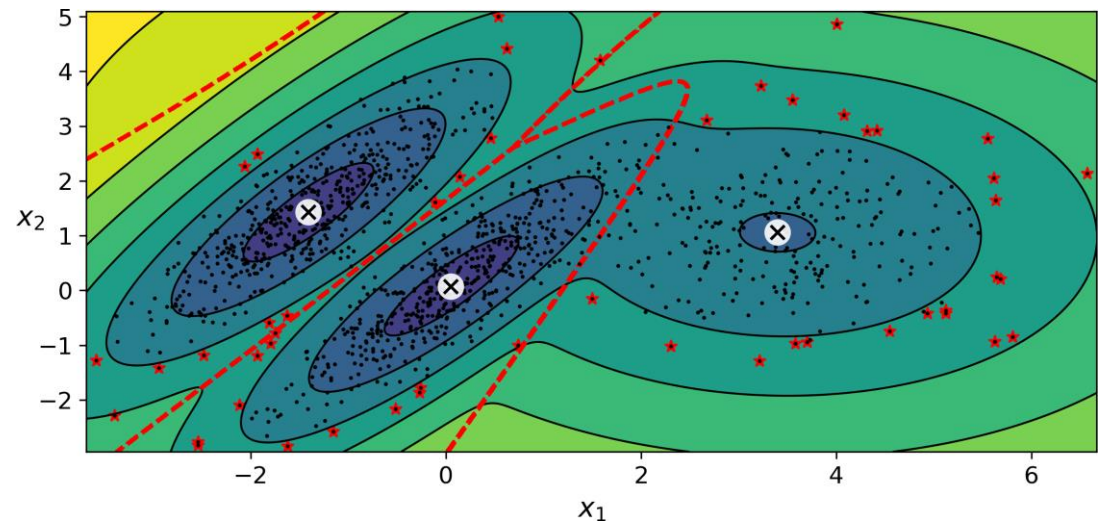
```
gm.covariances_
```

```
array([[[ 1.14807234, -0.03270354],  
        [-0.03270354,  0.95496237]],  
       [[ 0.63478101,  0.72969804],  
        [ 0.72969804,  1.1609872 ]],  
       [[ 0.68809572,  0.79608475],  
        [ 0.79608475,  1.21234145]])
```

Anomaly Detection using Gaussian Mixtures

- Any instance **located in a low-density region** can be considered an anomaly.
- **Identify the outliers** using the **4th percentile** lowest density as the threshold.

```
densities = gm.score_samples(X)
density_threshold = np.percentile(
    densities, 4)
anomalies = X[densities <
    density_threshold]
```



Selecting the Number of Components

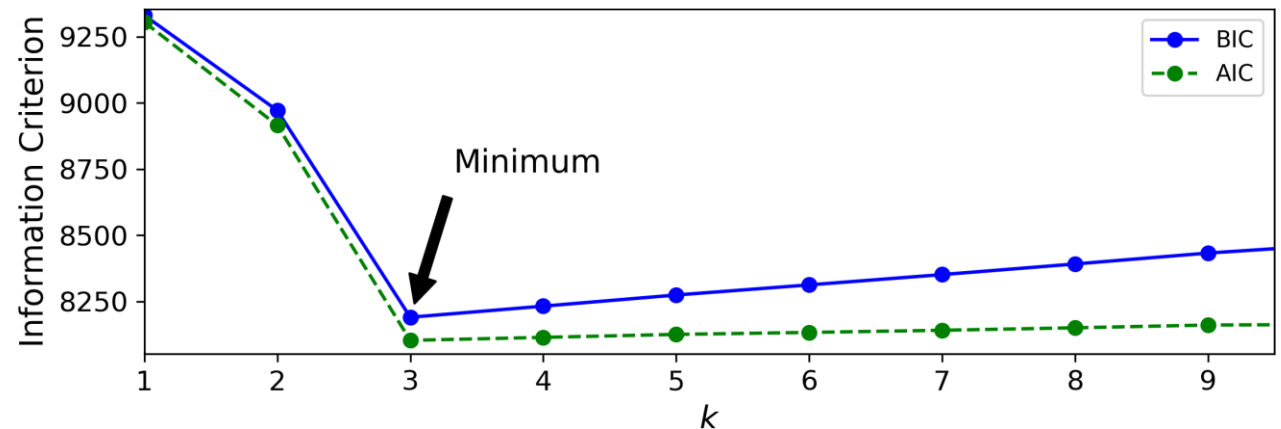
- Minimize the **Bayesian information criterion (BIC)** or the **Akaike information criterion (AIC)**.

`gm.bic(X)`

8189.74345832983

`gm.aic(X)`

8102.518178214792



Outline

- Dimensionality Reduction
 - Projection and Manifold
 - Principal Component Analysis (PCA)
- Unsupervised Learning
- Clustering
 - K-Means
 - DBSCAN
- Gaussian Mixtures and Anomaly Detection
- Exercises

Exercises

8.9. Load the **MNIST dataset** (introduced in Chapter 3) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use **PCA** to reduce the dataset's dimensionality, with an explained variance ratio of **95%**. Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next evaluate the classifier on the test set: how does it compare to the previous classifier?

Exercises

9.3. Describe two techniques to **select the right number of clusters** when using **K-Means**.

Exercises

9.10. The classic **Olivetti faces dataset** contains 400 grayscale 64×64 -pixel images of faces. Each image is flattened to a 1D vector of size 4,096. 40 different people were photographed (10 times each), and the usual task is to train a model that can predict which person is represented in each picture. Load the dataset using the **`sklearn.datasets.fetch_olivetti_faces()`** function, then split it into a training set, a validation set, and a test set (note that the dataset is already scaled between 0 and 1). Since the dataset is quite small, you probably want to use stratified sampling to ensure that there are the same number of images per person in each set. Next, **cluster the images** using **KMeans**, and ensure that you have a good number of clusters (using one of the techniques discussed in this chapter). Visualize the clusters: do you see similar faces in each cluster?

Summary

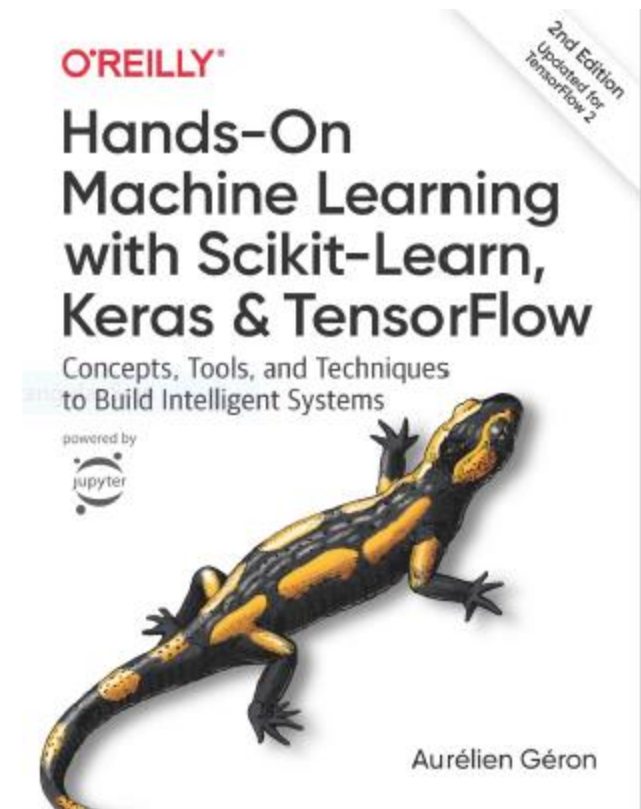
- Dimensionality Reduction
 - Projection and Manifold
 - Principal Component Analysis (PCA)
- Unsupervised Learning
- Clustering
 - K-Means
 - DBSCAN
- Gaussian Mixtures and Anomaly Detection
- Exercises

Neural Networks

Prof. Gheith Abandah

Reference

- Chapter 10: **Introduction to Artificial Neural Networks with Keras**
- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>



Introduction

- YouTube Video: *But what *is* a Neural Network?* from 3Blue1Brown

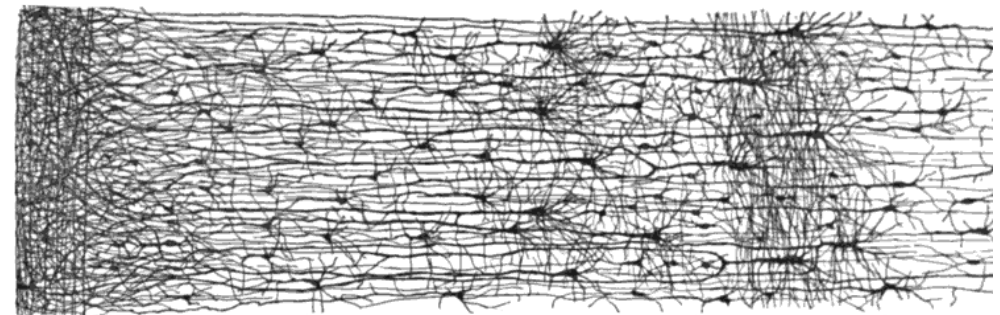
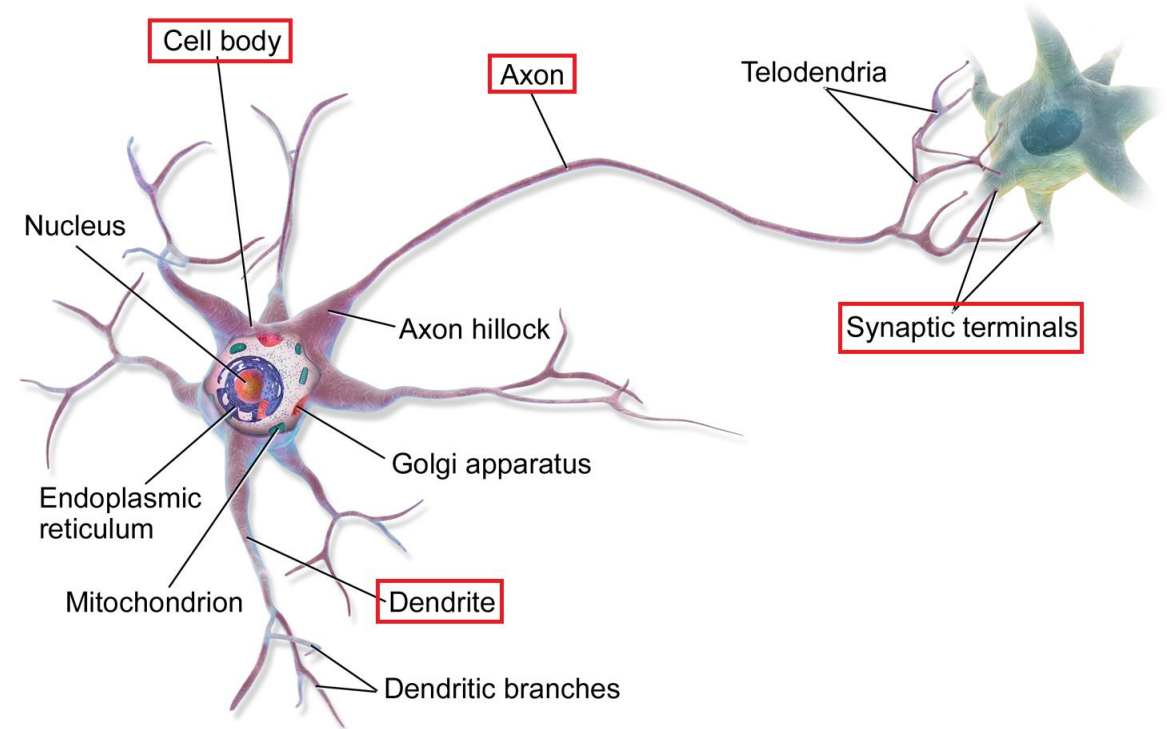
<https://youtu.be/aircAruvnKk>

Outline

1. Introduction
2. The perceptron
3. Multi-layer perceptron (MLP)
4. Regression MLPs
5. Classification MLPs

1. Introduction

- **Artificial neural networks** (ANNs) are inspired by the brain's architecture.
- First suggested in 1943. Is now **flourishing** due to the availability of:
 - Data
 - Computing power
 - Better algorithms



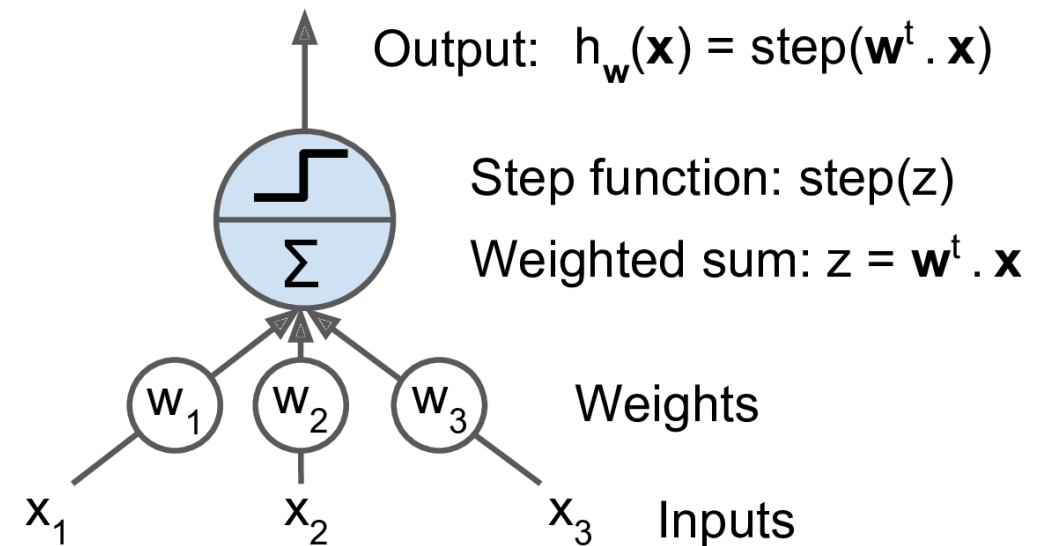
2. The Perceptron

- The **Perceptron** is a simple ANN, invented in 1957 and can perform linear binary classification or regression.

- Common **step function**:

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

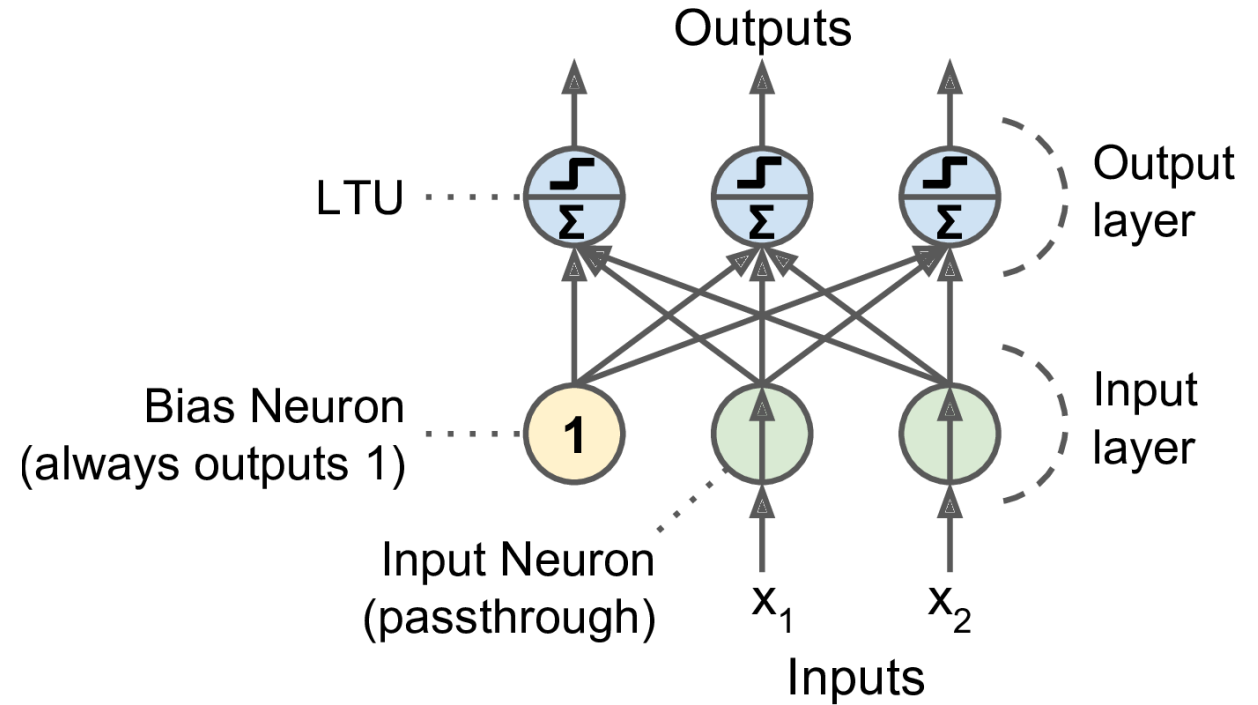
Linear threshold unit (LTU)



2. The Perceptron

- The Perceptron has an **input layer** with **bias** and **output layer**.
- With **multiple output nodes**, it can perform multiclass classification.
- Hebbian learning “**Cells that fire together, wire together.**”

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$



2. The Perceptron

- Scikit-Learn provides a **Perceptron** class.

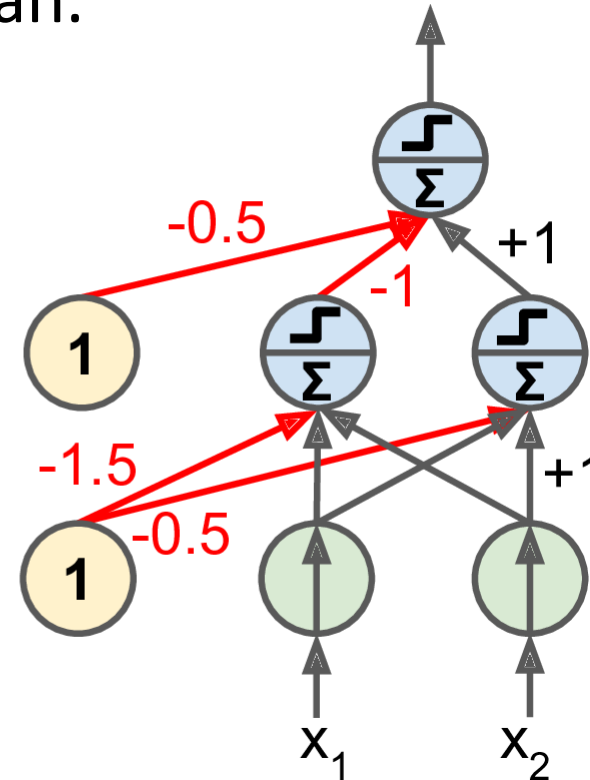
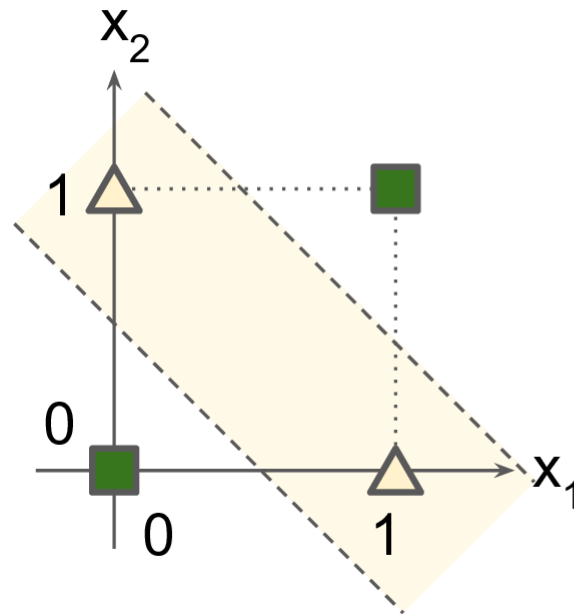
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

2. The Perceptron

- The perceptron **cannot solve non-linear problems** such as the XOR problem.
- The **Multi-Layer Perceptron** (MLP) can.

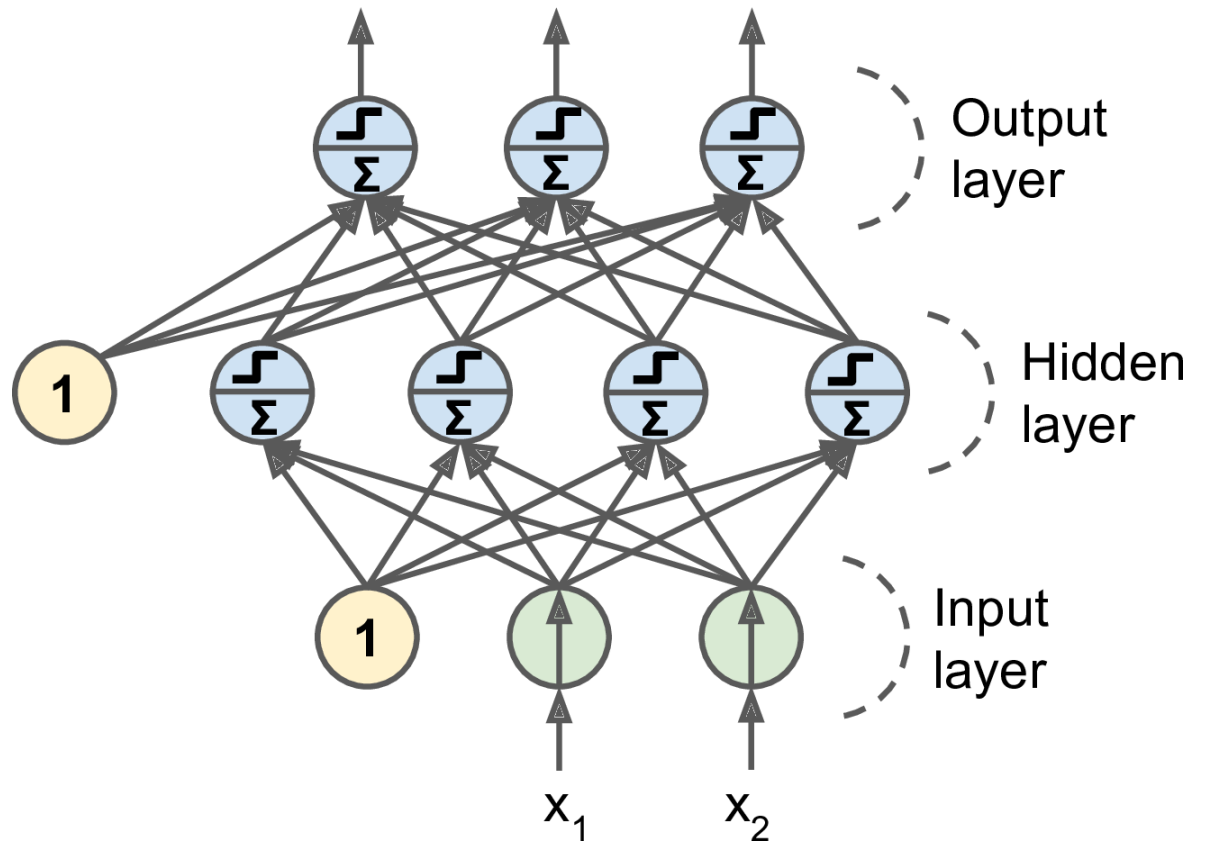


Outline

1. Introduction
2. The perceptron
3. Multi-layer perceptron (MLP)
4. Regression MLPs
5. Classification MLPs

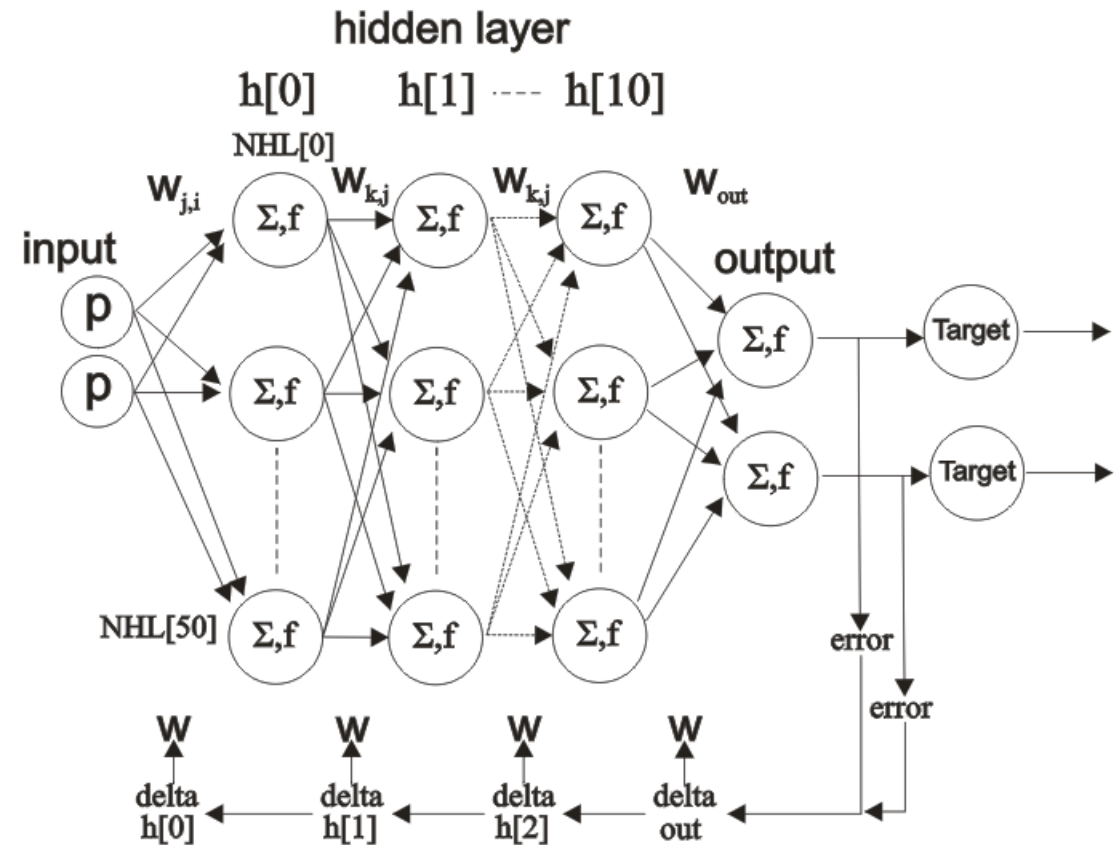
3. Multi-Layer Perceptron (MLP)

- An MLP is composed of a (pass-through) **input layer**, one or more layers of LTUs, called **hidden layers**, and a final layer of LTUs called the **output layer**.
- When an ANN has **two or more** hidden layers, it is called a **deep neural network (DNN)**.



3. Multi-Layer Perceptron (MLP)

- Trained using the **backpropagation training algorithm**.
 - For each training instance the algorithm first makes a prediction (**forward pass**), measures the error,
 - then goes through each layer in reverse to measure the error contribution from each connection (**reverse pass**),
 - and finally slightly tweaks the connection weights to reduce the error (**Gradient Descent step**).



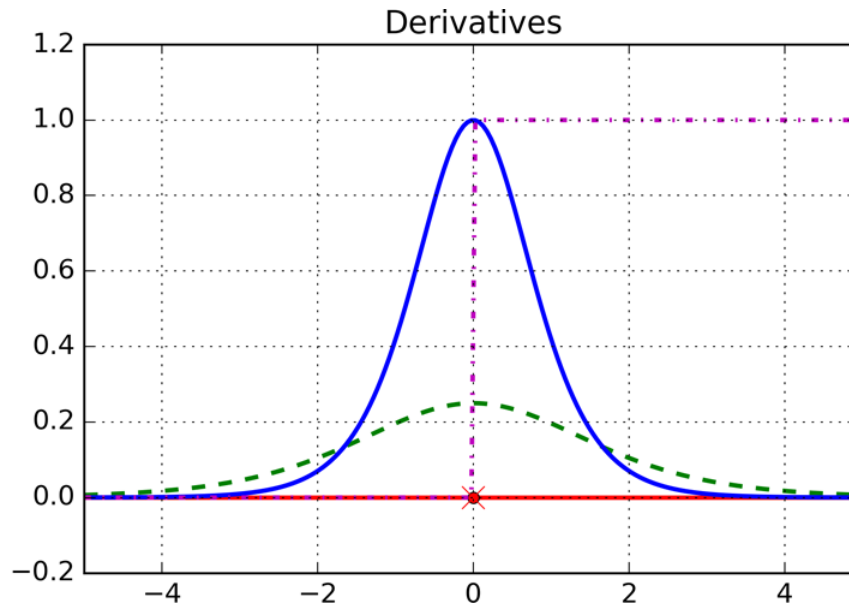
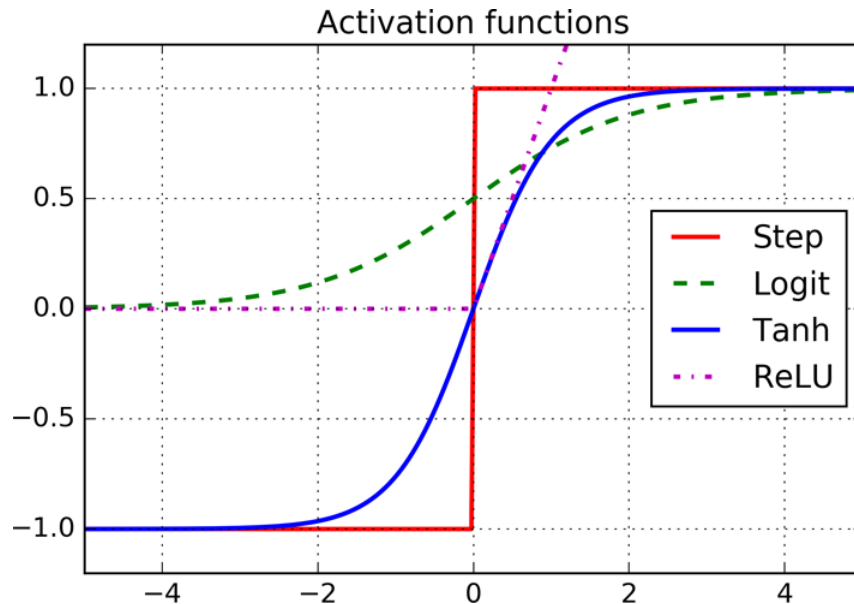
3. Multi-Layer Perceptron (MLP)

- **Common activation functions: logistic, hyperbolic tangent, and rectified linear unit.**

$$\sigma(z) = 1 / (1 + \exp(-z))$$

$$\tanh(z) = 2\sigma(2z) - 1$$

$$\text{ReLU}(z) = \max(0, z)$$



Outline

1. Introduction
2. The perceptron
3. Multi-layer perceptron (MLP)
4. Regression MLPs
5. Classification MLPs

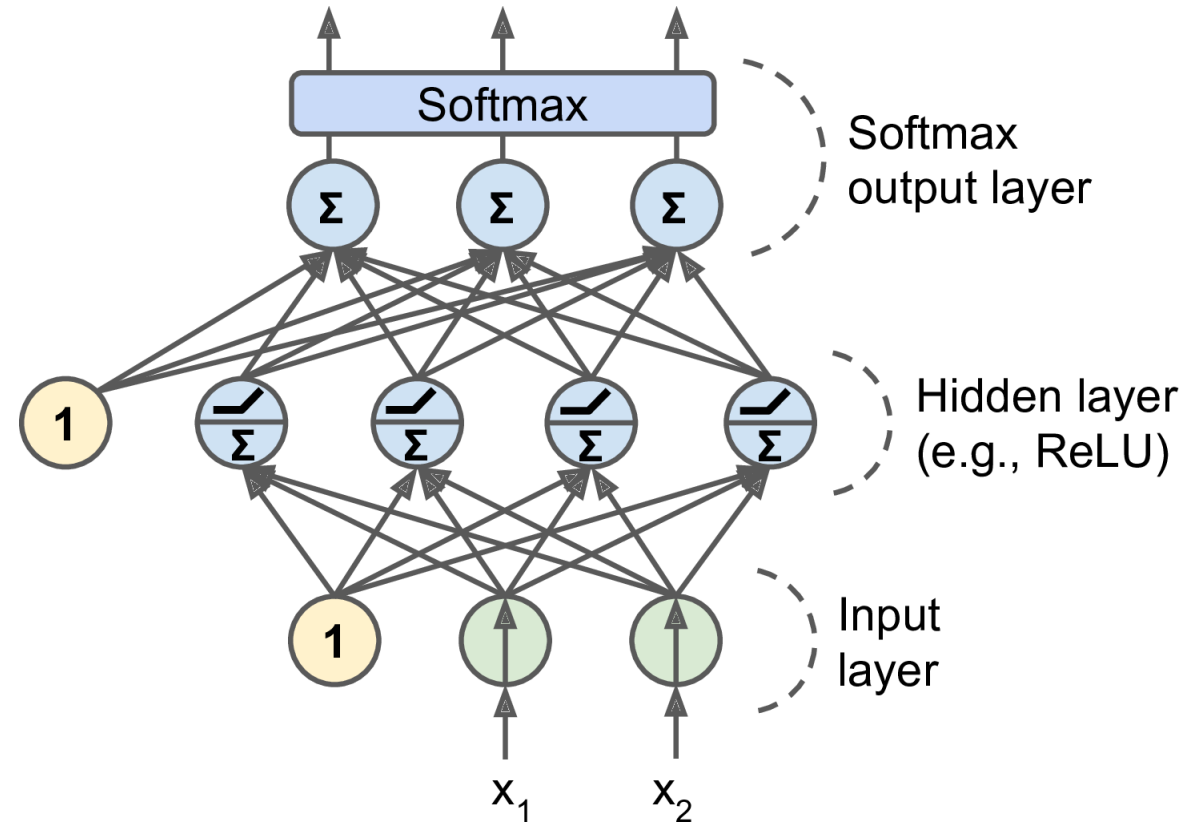
4. Regression MLPs

- Typical MLP architecture for **regression**:

Hyperparameter	Typical Value
# input neurons	One per input feature (e.g., $28 \times 28 = 784$ for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

5. Classification MLPs

- For **classification**, the output layer uses the **softmax function**.
- The output of each neuron corresponds to the **estimated probability** of the corresponding class.



$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k$$

5. Classification MLPs

- Typical MLP architecture for **classification**:

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross-Entropy	Cross-Entropy	Cross-Entropy

Summary

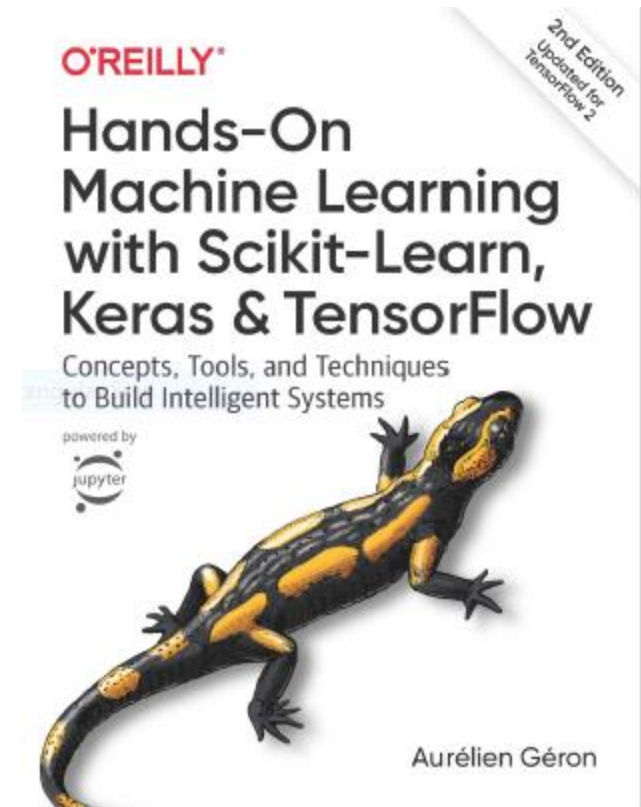
1. Introduction
2. The perceptron
3. Multi-layer perceptron (MLP)
4. Regression MLPs
5. Classification MLPs

Artificial Neural Networks with Keras

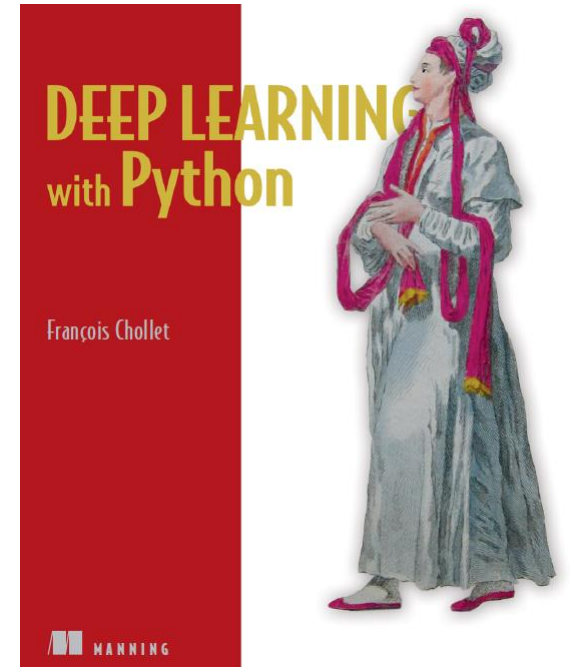
Prof. Gheith Abandah

Reference

- Chapter 10: **Introduction to Artificial Neural Networks with Keras**
- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>



Reference



- **Deep Learning with Python**, by François Chollet, Manning Pub. 2018
- **Introduction to Keras** by François Chollet, March 9th, 2018 ([slides](#))

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

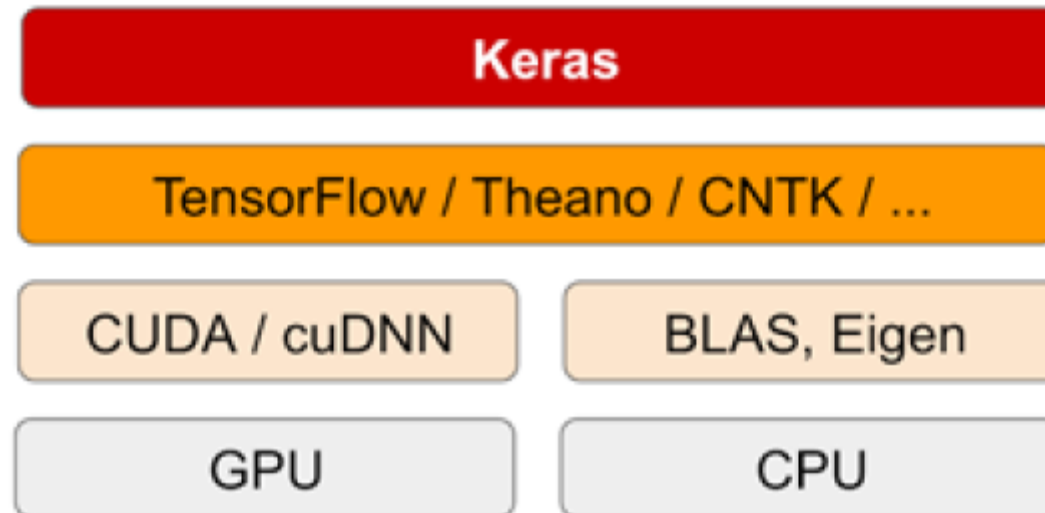
Introduction

- YouTube Video: *Keras Explained* from Siraj Raval

https://youtu.be/j_pJmXJwMLA

1. Introduction

- **Keras** is a high-level API to build and train deep learning models.



1. Introduction – Advantages

- **User friendly**: Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.
- **Modular and composable**: Keras models are made by connecting configurable building blocks together, with few restrictions.
- **Easy to extend**: Write custom building blocks to express new ideas for research. Create new layers, loss functions, and develop state-of-the-art models.

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

2. Keras API Styles

1. The Sequential Model

- Dead simple
- Only for single-input, single-output, sequential layer stacks
- Good for 70+% of use cases

2. The functional API

- Like playing with Lego bricks
- Multi-input, multi-output, arbitrary static graph topologies
- Good for 95% of use cases

3. Model subclassing

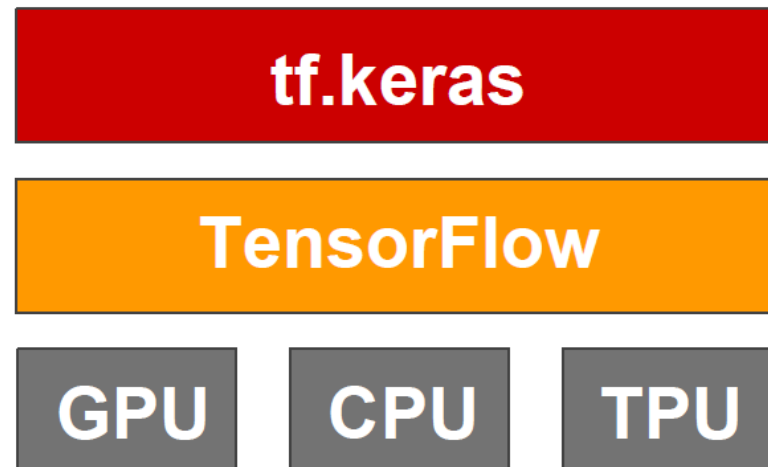
- Maximum flexibility
- Larger potential error surface

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

3. TensorFlow Keras

- Keras is the official high-level API of TensorFlow
- tensorflow.keras (tf.keras) module
- Part of core TensorFlow since v1.4
- Full Keras API
- With useful extra features such as **tf.data**



3. TensorFlow Keras

- To install TensorFlow

```
$ pip install --upgrade tensorflow
```

- To import Keras from TensorFlow

```
>>> import tensorflow as tf
```

```
>>> from tensorflow.keras import Layers
```


```
>>> from tensorflow import keras
```

```
>>> tf.__version__
```

```
'2.1.0'
```

```
>>> keras.__version__
```

```
'2.2.4-tf'
```

- 
- Dense
 - Activations
 - Dropout
 - Conv1D, 2D, 3D
 - Pooling
 - RNN, LSTM, GRU
 - ...

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

4. Image Classifier Using the Sequential Model

- **Fashion MNIST** is similar to MNIST (70,000 grayscale images of 28x28 pixels each, with 10 classes).



4. Fashion MNIST

1. **Get and prepare the dataset.**
2. **Build sequential model** of layers that maps your inputs to your targets.
3. **Compile the model and configure the learning process** by choosing a loss function, an optimizer, and some metrics to monitor.
4. **Train the model** by calling the `fit()` method of your model.
5. **Evaluate and use** the model.

4.1 Get and Prepare the Dataset

```
import tensorflow as tf
from tensorflow import keras

# Get the Fashion MNIST
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) =
    fashion_mnist.load_data()

# Prepare the data train (55000), val (5000), test (10000)
X_valid = X_train_full[:5000] / 255.
X_train = X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```


4.2 Build the Model

The default is no activation function, i.e., linear layer.

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

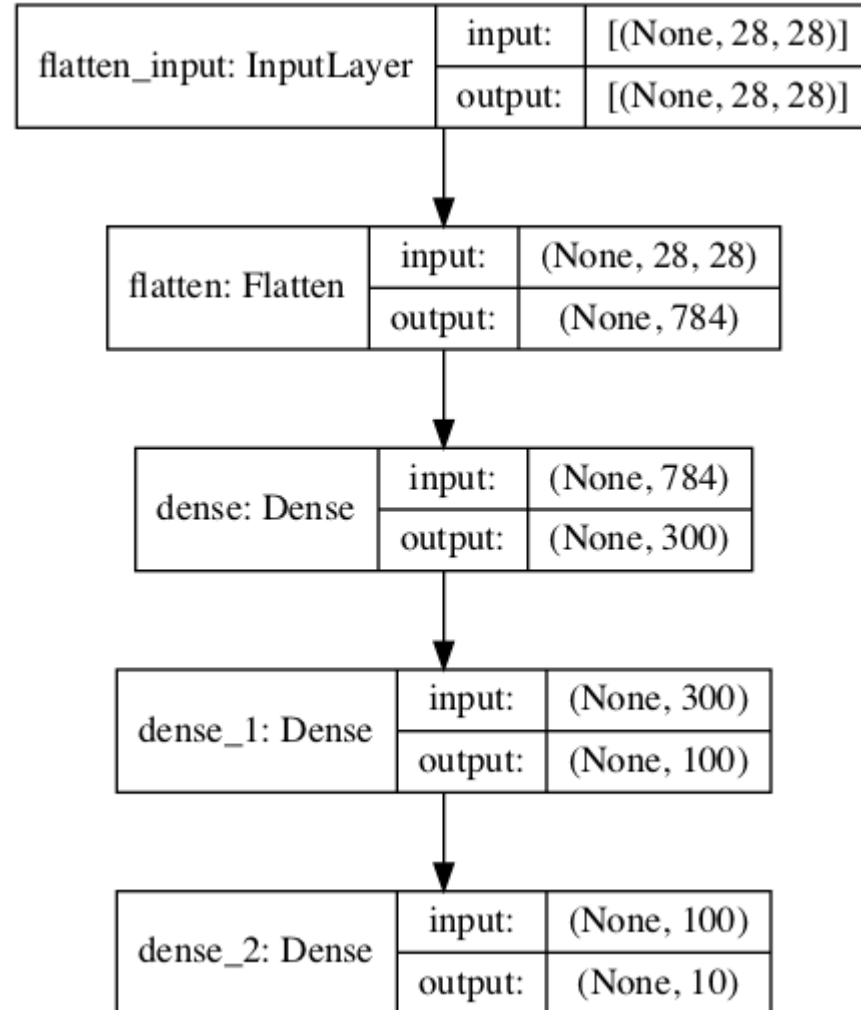
```
>>> model.summary()
```

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_3 (Dense)	(None, 300)	235500
dense_4 (Dense)	(None, 100)	30100
dense_5 (Dense)	(None, 10)	1010

=====
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0

4.2 Build the Model

```
# Plot the model
keras.utils.plot_model(
    model,
    "my_model.png",
    show_shapes=True)
```



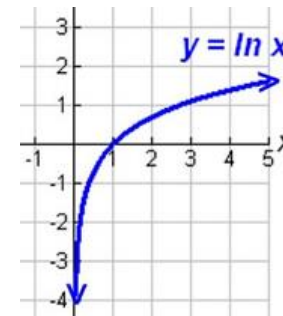
4.3 Compile the Model

```
model.compile(loss="sparse_categorical_crossentropy",  
              optimizer="sgd",  
              metrics=["accuracy"])
```

Stochastic Gradient
Descent

```
# For sparse labels (0-9):  
loss = "sparse_categorical_crossentropy"  
# For one-hot labels:  
loss = "categorical_crossentropy"  
# For binary labels:  
loss = "binary_crossentropy"  
# For regression:  
loss = "mean_squared_error"
```

$$-\sum_{c=1}^M y_{o,c} \log(p_{o,c})$$



4.4 Train the Model

Train the model

```
history = model.fit(X_train, y_train, epochs=30,  
                    validation_data=(X_valid, y_valid))
```

Train on 55000 samples, validate on 5000 samples

Epoch 1/30

55000/55000 [=====] - 2s 44us/sample - loss: 0.7226 - accuracy: 0.7641 - val_loss:
0.5073 - val_accuracy: 0.8320

Epoch 2/30

55000/55000 [=====] - 2s 39us/sample - loss: 0.4844 - accuracy: 0.8321 - val_loss:
0.4541 - val_accuracy: 0.8478

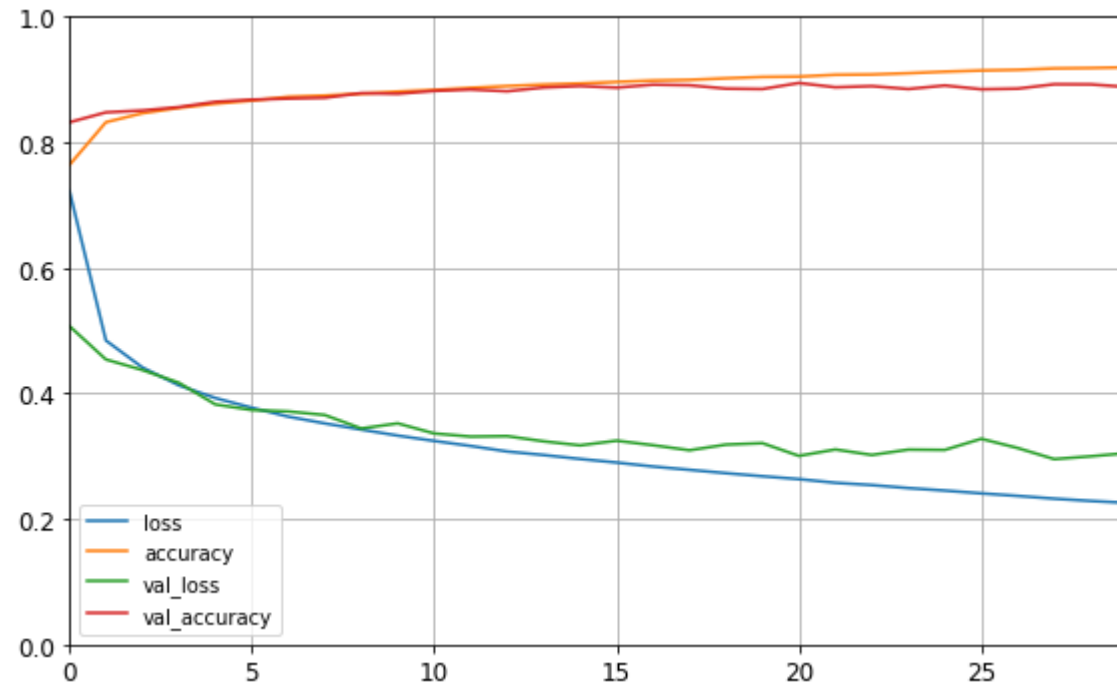
...

Epoch 30/30

55000/55000 [=====] - 2s 39us/sample - loss: 0.2256 - accuracy: 0.9195 - val_loss:
0.3049 - val_accuracy: 0.8882

4.4 Train the Model

```
import pandas as pd
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
save_fig("keras_learning_curves_plot")
plt.show()
```



4.5 Evaluate and Use the Model

```
model.evaluate(X_test, y_test)
10000/10000 [=====] - 0s 21us/sample - loss: 0.3378 -
accuracy: 0.8781
[0.33780701770782473, 0.8781]
```

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99],
       [0. , 0. , 0.99, 0. , 0.01, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
```

```
model.predict_classes(X_new)
array([9, 2, 1])
```

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

5. Example - MNIST

1. **Define your training data**: input tensors and target tensors.
2. **Define a network** of layers (or **model**) that maps your inputs to your targets.
3. **Configure the learning process** by choosing a loss function, an optimizer, and some metrics to monitor.
4. **Iterate on your training data** by calling the **fit()** method of your model.

5. Example – Prepare the data

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) =
    mnist.load_data()
 #(60000, 28, 28), (60000), #(10000, 28, 28), (10000)
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255

from keras.utils import to_categorical #one hot
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

5. Example – Define and configure the network

```
from keras import models
from keras import layers
```

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))

network.compile(optimizer='rmsprop',
               loss='categorical_crossentropy',
               metrics=['accuracy'])
```

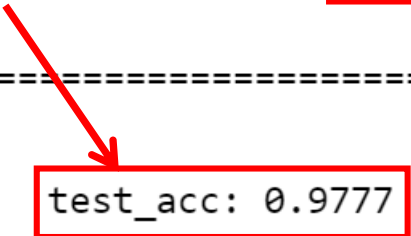
5. Example – Training and evaluation

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5  
60000/60000 [=====] - 2s - loss: 0.2577 - acc: 0.9245  
Epoch 2/5  
60000/60000 [=====] - 1s - loss: 0.1042 - acc: 0.9690  
Epoch 3/5  
60000/60000 [=====] - 1s - loss: 0.0687 - acc: 0.9793  
Epoch 4/5  
60000/60000 [=====] - 1s - loss: 0.0508 - acc: 0.9848  
Epoch 5/5  
60000/60000 [=====] - 1s - loss: 0.0382 - acc: 0.9890
```

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
```

```
9536/10000 [=====>..] - ETA: 0s
```



```
test_acc: 0.9777
```

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

6. Regression Using the Sequential Model

- Solve the **California housing** problem using a regression neural network.
- Scikit-Learn has **fetch_california_housing()** function to load the data
- This dataset contains **only numerical features** and there are **no missing values**.

6.1 Get and Prepare the Dataset

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
housing = fetch_california_housing()
```

The default is 75% : 25%



```
X_train_full, X_test, y_train_full, y_test =
    train_test_split(housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,
    y_train_full, random_state=42)
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

6.2 Build and Compile the Model

```
# Building by passing a list of layers when creating  
# the Sequential model
```

```
model = keras.models.Sequential(  
    keras.layers.Dense(30, activation="relu",  
        input_shape=X_train.shape[1:]),  
    keras.layers.Dense(1)  
])
```

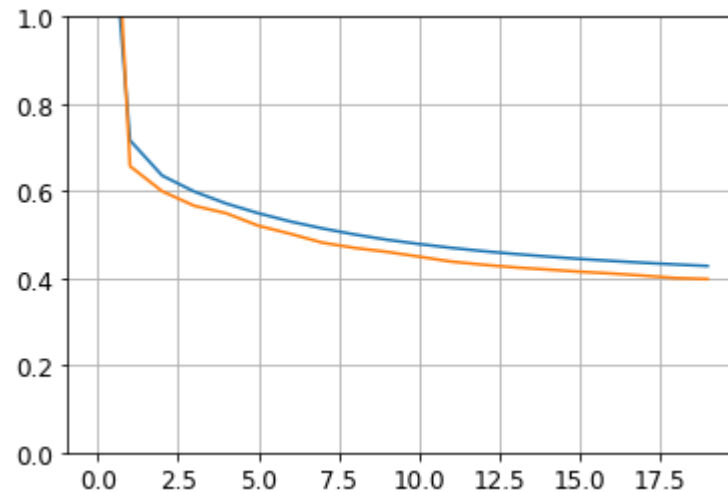
The default is 0.01



```
# Compile with creating an optimizer object  
model.compile(loss="mean_squared_error",  
    optimizer=keras.optimizers.SGD(lr=1e-3))
```

6.3 Train and Evaluate the Model

```
history = model.fit(X_train, y_train, epochs=20,  
                    validation_data=(X_valid, y_valid))
```



```
mse_test = model.evaluate(X_test, y_test)  
5160/5160 [=====] - 0s  
15us/sample - loss: 0.421
```


6.4 Save and Restore the Model

- After training a model save it to a file.

```
model.save("my_keras_model.h5")
```

- In the production program, load the trained model.

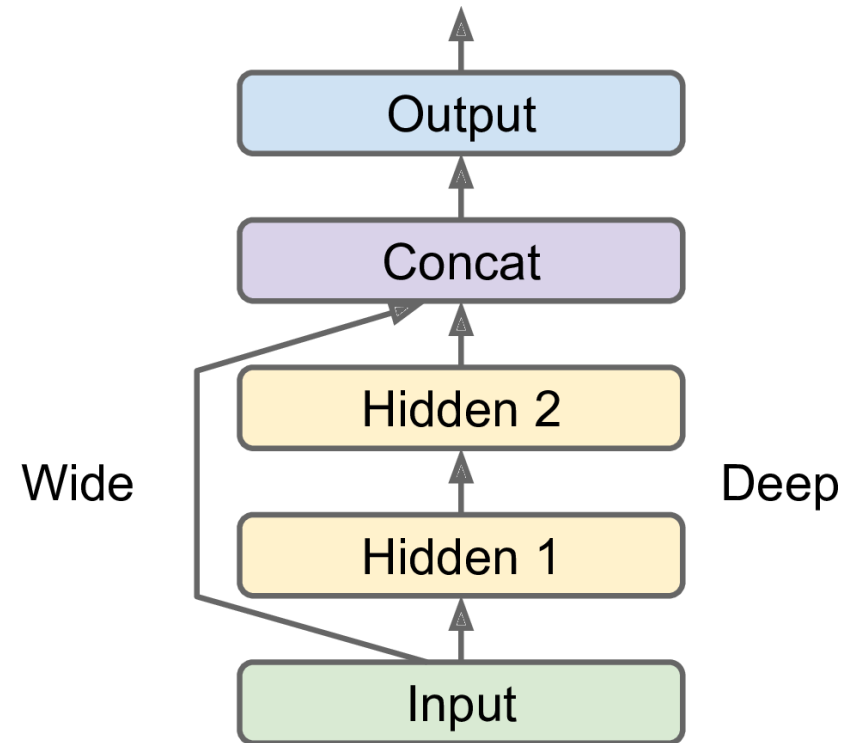
```
model = keras.models.load_model("my_keras_model.h5")
```

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

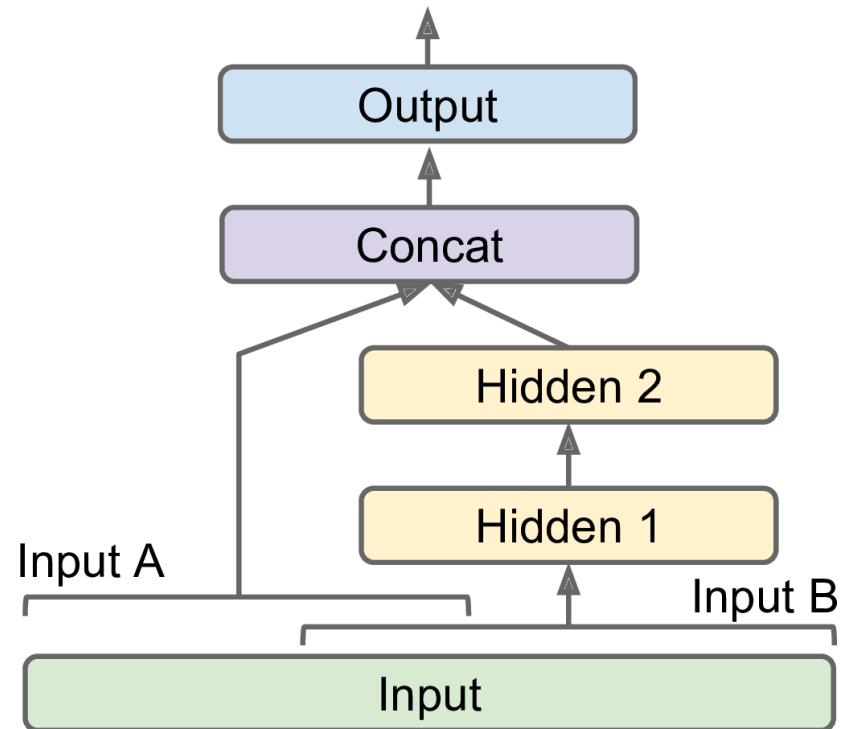
7. Using the Functional API

- Keras functional API can be used to build arbitrary **static graph topologies**.
- Create a layer and as soon as it is created, **call it like a function**, passing it the input.
- Example 1: the **wide and deep** network that learns both deep patterns (using the deep path) and simple rules (through the short path).



7. Using the Functional API

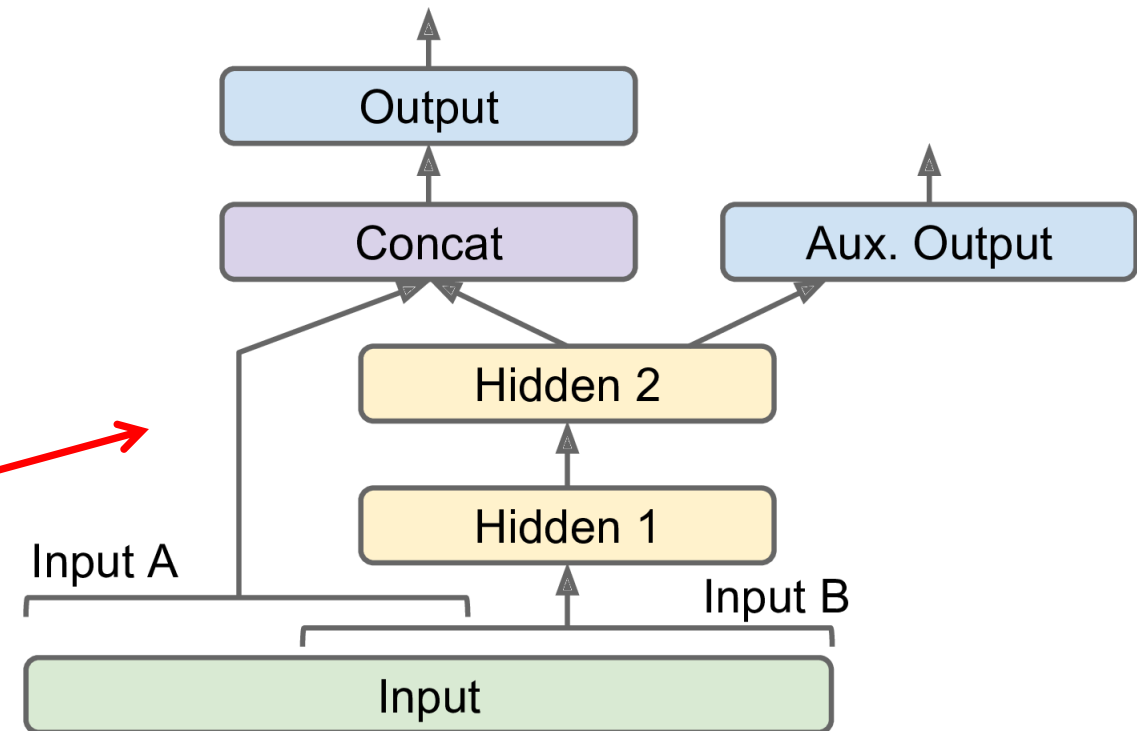
- 2. Multi-input:** You can send a subset of the features through the wide path, and a different subset (possibly overlapping) through the deep path.



7. Using the Functional API

3. Multiple Outputs

- To **locate and classify** the main object in a picture.
- **Multiple independent tasks** to perform based on the same data.
- **Regularization technique** (to ensure that the deep network learns something useful on its own).



7.1 Auxiliary Output for Regularization

```
# Build the model
```

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
```

```
input_B = keras.layers.Input(shape=[6], name="deep_input")
```

```
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
```

```
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
```

```
concat = keras.layers.concatenate([input_A, hidden2])
```

```
output = keras.layers.Dense(1, name="main_output")(concat)
```

```
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
```

```
model = keras.models.Model(inputs=[input_A, input_B],  
                             outputs=[output, aux_output])
```

7.1 Auxiliary Output for Regularization

```
# Split the input
```

```
X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
```

```
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
```

```
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
```

```
# Take some test samples
```

```
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]
```

7.1 Auxiliary Output for Regularization

```
# Compile, train, evaluate, and predict
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1],
              optimizer=keras.optimizers.SGD(lr=1e-3))

history = model.fit([X_train_A, X_train_B], [y_train, y_train], epochs=20,
                  validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))

total_loss, main_loss, aux_loss = model.evaluate([X_test_A, X_test_B],
                                                [y_test, y_test])

y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```


Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

8. Using Callbacks

- The **fit()** method accepts a **callbacks** argument that lets you specify a list of objects that Keras will call during training
 - at the start and end of **training**
 - at the start and end of each **epoch**
 - before and after processing each **batch**
- There are many callbacks available in the **keras.callbacks** package. See

<https://keras.io/callbacks/>

8.1 Saving Best Model

- **Save your best model** when its performance on the validation set is the best so far.

```
checkpoint_cb = keras.callbacks.ModelCheckpoint(
    "my_keras_model.h5", save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
    validation_data=(X_valid, y_valid),
    callbacks=[checkpoint_cb])
```

```
# rollback to best model
model = keras.models.load_model("my_keras_model.h5")
mse_test = model.evaluate(X_test, y_test)
```

8.2 Early Stopping

- Interrupt training when there is no progress on the validation set for a number of epochs (defined by the **patience** argument)
- Optionally roll back to the best model.

```
early_stopping_cb = keras.callbacks.EarlyStopping(  
    patience=10, restore_best_weights=True)
```

```
history = model.fit(X_train, y_train, epochs=100,  
    validation_data=(X_valid, y_valid),  
    callbacks=[checkpoint_cb, early_stopping_cb])
```

Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

9. Visualization Using TensorBoard

- TensorBoard is a great **interactive visualization tool** that comes with TensorFlow.
- Use it using its callback

```
tensorboard_cb =
```

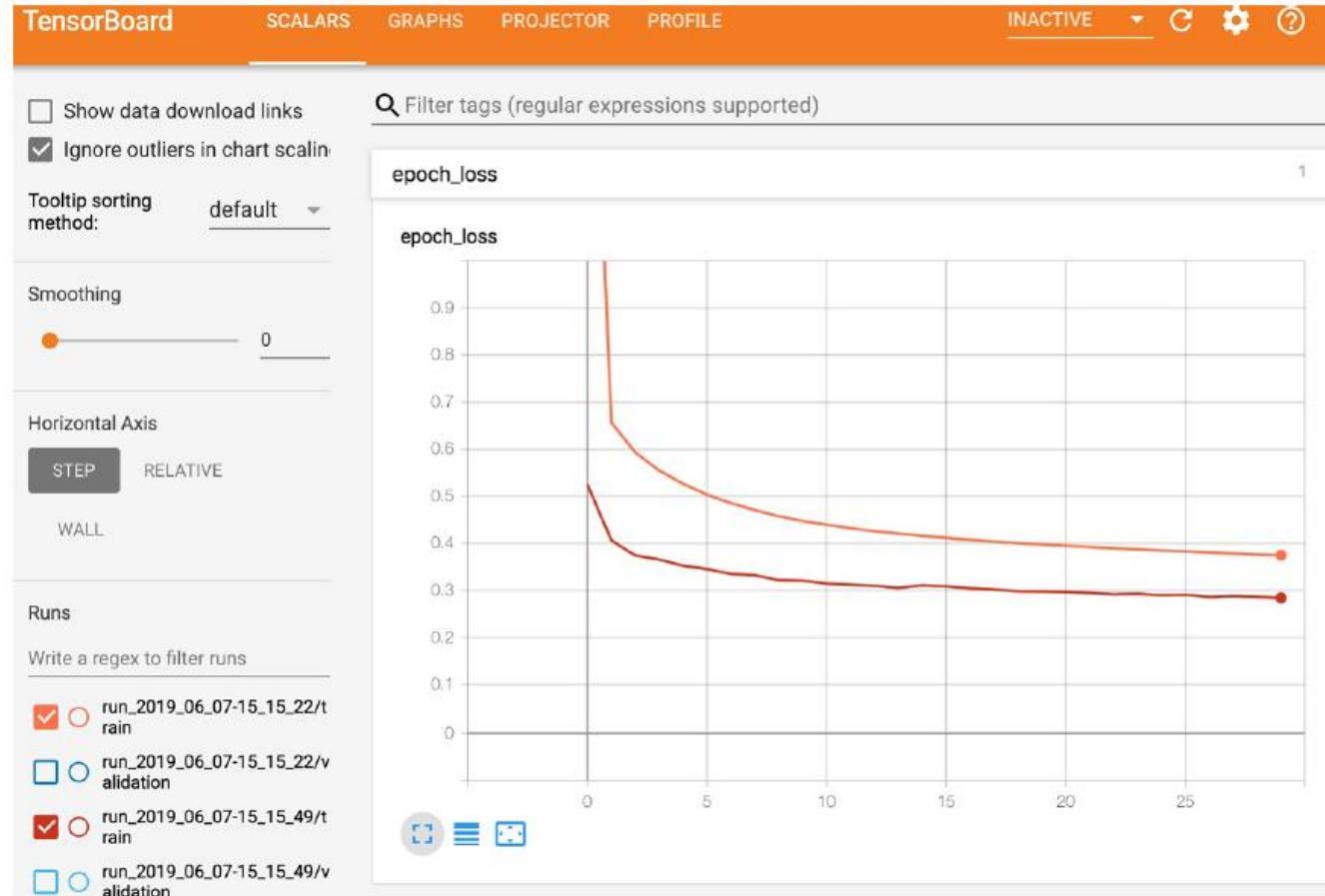
```
    keras.callbacks.TensorBoard(run_logdir)
```

```
history = model.fit(X_train, y_train, epochs=30,  
                    validation_data=(X_valid, y_valid),  
                    callbacks=[tensorboard_cb])
```

- Start TensorBoard server

```
$ tensorboard --logdir=./my_logs --port=6006
```

9. Open <http://localhost:6006>



Outline

1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

10. Fine-Tuning Neural Network Hyperparameters

- **Number of Hidden Layers**
 - One hidden layer can theoretically model even the most complex functions, provided it has enough neurons.
 - But for complex problems, deep networks have a much higher parameter efficiency than shallow ones.
- **Number of Neurons per Hidden Layer**
 - **Pyramid** across layers or **same** size
 - **Stretch pants**: pick a model with more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting.
- Better to increase the number of layers instead of the number of neurons per layer.

10. Fine-Tuning Neural Network Hyperparameters

- **Learning Rate**: the optimal LR is about half of the maximum LR.
- **Optimizer**: There are other than the Mini-batch Gradient Descent optimizer.
- **Batch Size**
 - Larger gives better speed up with hardware accelerators.
 - Smaller makes the models more general.
- **Activation Functions**

11. Tutorials

- <https://keras.io/>
- <https://www.tensorflow.org/guide/keras>
- Keras Tutorial: Deep Learning in Python from DataCamp, <https://www.datacamp.com/community/tutorials/deep-learning-python>
- Keras Tutorial: The Ultimate Beginner's Guide to Deep Learning in Python, from EliteDataScience, <https://elitedatascience.com/keras-tutorial-deep-learning-in-python>

12. Exercise

From Chapter 10, solve exercise:

- 10. Train a deep MLP on the **MNIST** dataset (you can load it using `keras.datasets.mnist.load_data()`). See if you can get over **98%** precision. Try searching for the optimal learning rate by using the approach presented in this chapter (i.e., by growing the learning rate exponentially, plotting the error, and finding the point where the error shoots up). Try adding all the bells and whistles—save checkpoints, use **early stopping**, and plot learning curves using **TensorBoard**.

Summary

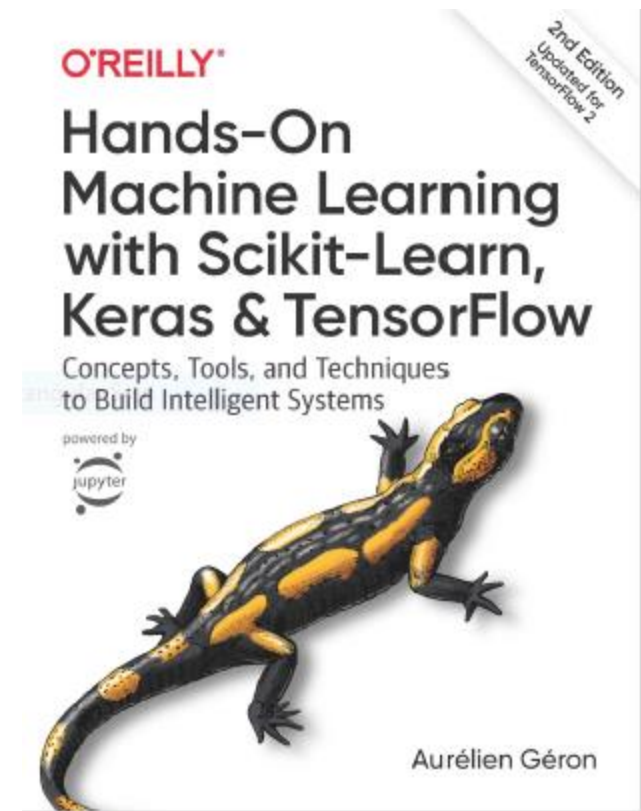
1. Introduction
2. Keras API Styles
3. TensorFlow Keras
4. Image Classifier Using the Sequential Model
5. Example - MNIST
6. Regression Using the Sequential Model
7. Using the Functional API
8. Using Callbacks
9. Visualization Using TensorBoard
10. Fine-Tuning Neural Network Hyperparameters
11. Tutorials
12. Exercise

Deep Neural Networks

Prof. Gheith Abandah

Reference

- Chapter 11: **Training Deep Neural Networks**



- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

1. Introduction

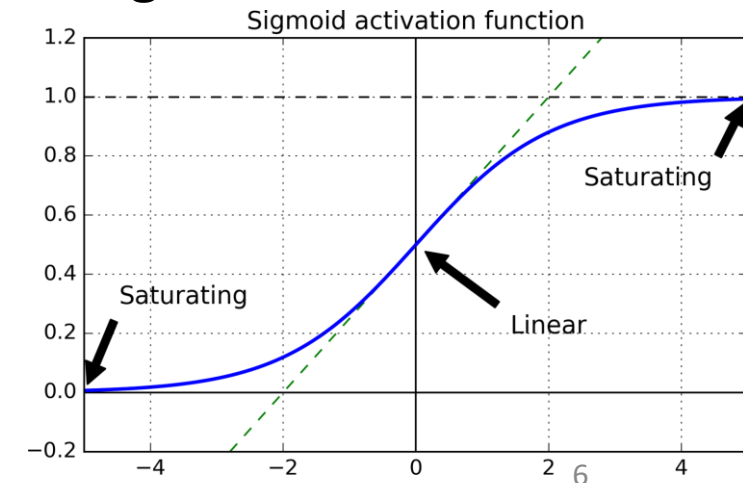
- Deep neural networks can solve **complex problems** and provide **end-to-end** solutions.
- When you train a deep network, you may face the following **problems**:
 - **Vanishing** or **exploding gradients**: The gradients grow smaller and smaller, or larger and larger.
 - **Not enough data**
 - **Long training time**
 - **Overfitting**

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

2. Vanishing/Exploding Gradients Problems

- **Vanishing Problem:** In the backpropagation algorithm, gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
 - Lower layers' connections are left unchanged.
- **Exploding Problem:** the gradients can grow bigger and bigger.
 - Layers get very large weight updates and the algorithm diverges.
- **Main Reasons:** Using activation functions (logistic sigmoid) and weight initialization (normal distribution with 0-mean and 1-standard deviation).



2.1 Glorot and He Initialization

- **Glorot and Bengio**: In order for the signal not to die out, nor to explode and saturate, the variance of the outputs of each layer should be equal to the variance of its inputs.
- **Solution**: the connection weights of each layer must be initialized randomly as follows:

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{avg}}}$

$$fan_{avg} = (fan_{in} + fan_{out})/2.$$

2.1 Glorot and He Initialization

- **Recommended** initialization parameters for each type of activation function.

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

- For the uniform distribution, use $r = \sqrt{3\sigma^2}$
- Keras uses **Glorot initialization** with a **uniform** distribution.

2.1 Glorot and He Initialization

- To change it to **He initialization**:

```
keras.layers.Dense(10, activation="relu",  
    kernel_initializer="he_normal") # Or "he_uniform"
```

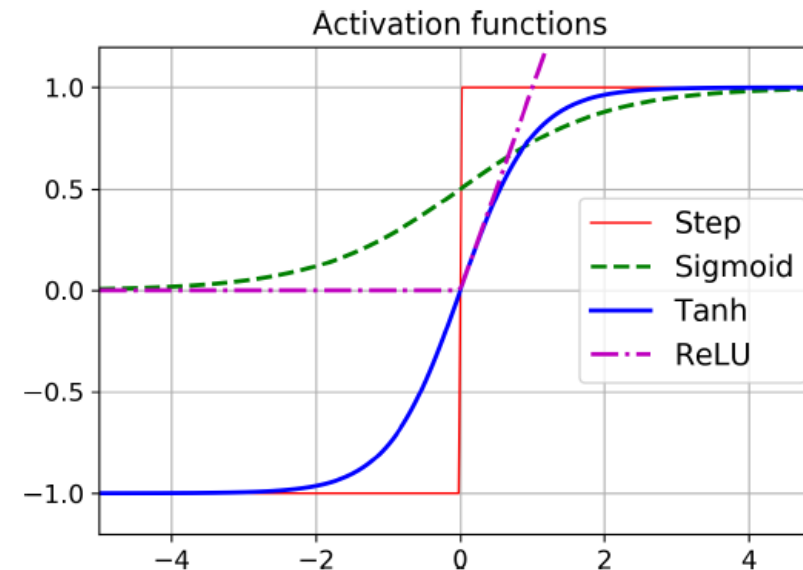
- **He initialization** with a **uniform** distribution but based on **fan_{avg}**:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2.,  
    mode='fan_avg', distribution='uniform')
```

```
keras.layers.Dense(10, activation="sigmoid",  
    kernel_initializer=he_avg_init)
```

2.2 Nonsaturating Activation Functions

- **Step** does not work with the back propagation algorithm.
- **ReLU** is better than **sigmoid** because it does not saturate for positive values and is fast.
- **Dying ReLUs**: A neuron dies when its input is negative for all training instances.

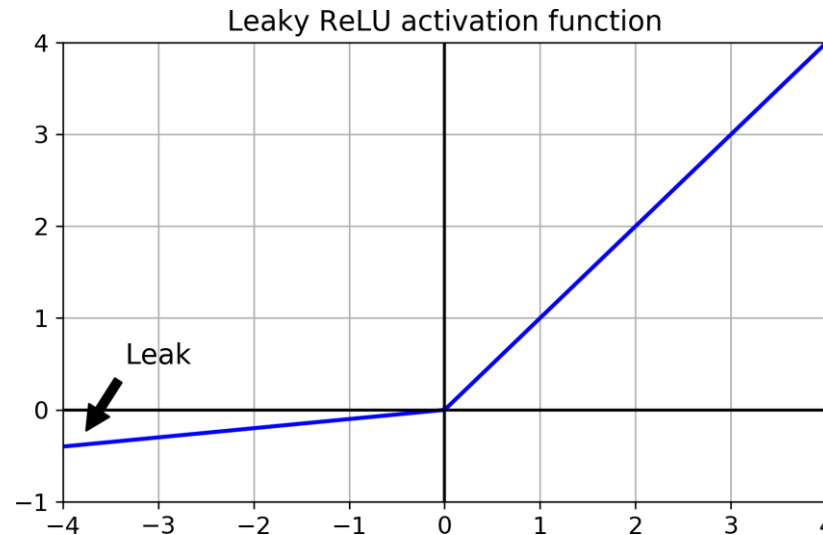


2.2 Nonsaturating Activation Functions

- **Leaky ReLU** performs better than ReLU.

$$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$$

- α between 0.01 and 0.3

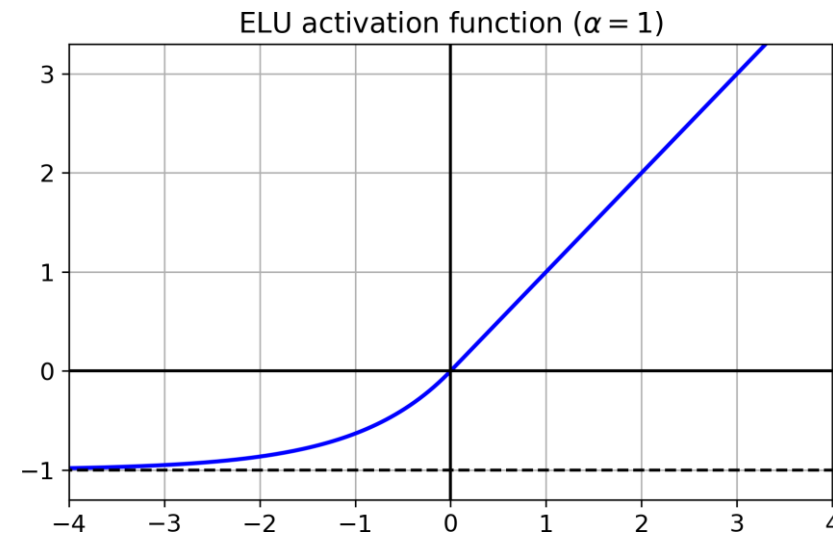


```
model = keras.models.Sequential([  
    ...  
    keras.layers.Dense(10, kernel_initializer="he_normal"),  
    keras.layers.LeakyReLU(alpha=0.2), # added as a layer  
    ...  
])
```


2.2 Nonsaturating Activation Functions

- **Exponential linear unit (ELU)** also performs better than ReLU but is slower.
- **Scaled ELU (SELU)** performs best with dense and CNN, but must scale inputs and use `lecun_normal`.

$$\text{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$



```
layer = keras.layers.Dense(10, activation="selu",  
                             kernel_initializer="lecun_normal")
```

2.2 Nonsaturating Activation Functions

- **Summary:**

- SELU > ELU > leaky ReLU > ReLU > tanh > logistic
- If you cannot use SELU, use ELU.
- For fast response, use leaky ReLU or ReLU.

2.3 Batch Normalization

- The techniques in §2.1 and §2.2 can significantly reduce the vanishing/exploding gradients problems at the **beginning of training**, but don't guarantee that they won't **come back during training**.
- **Batch Normalization (BN)** zero-centers and normalizes each layer input using statistics from the mini batch (> 30).
- **Other benefits**: Works even without §2.1 and §2.2, allows using larger LR, and have regularization effect.

2.3 Batch Normalization

- Implementing batch normalization with Keras is easy.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu",
        kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu",
        kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

2.4 Gradient Clipping

- Mitigates the exploding gradients problem by **clipping the gradients** during backpropagation so that they never exceed some threshold.
- Use it when you observe that the gradients are exploding during training. You can **track the size of the gradients** using TensorBoard.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
```

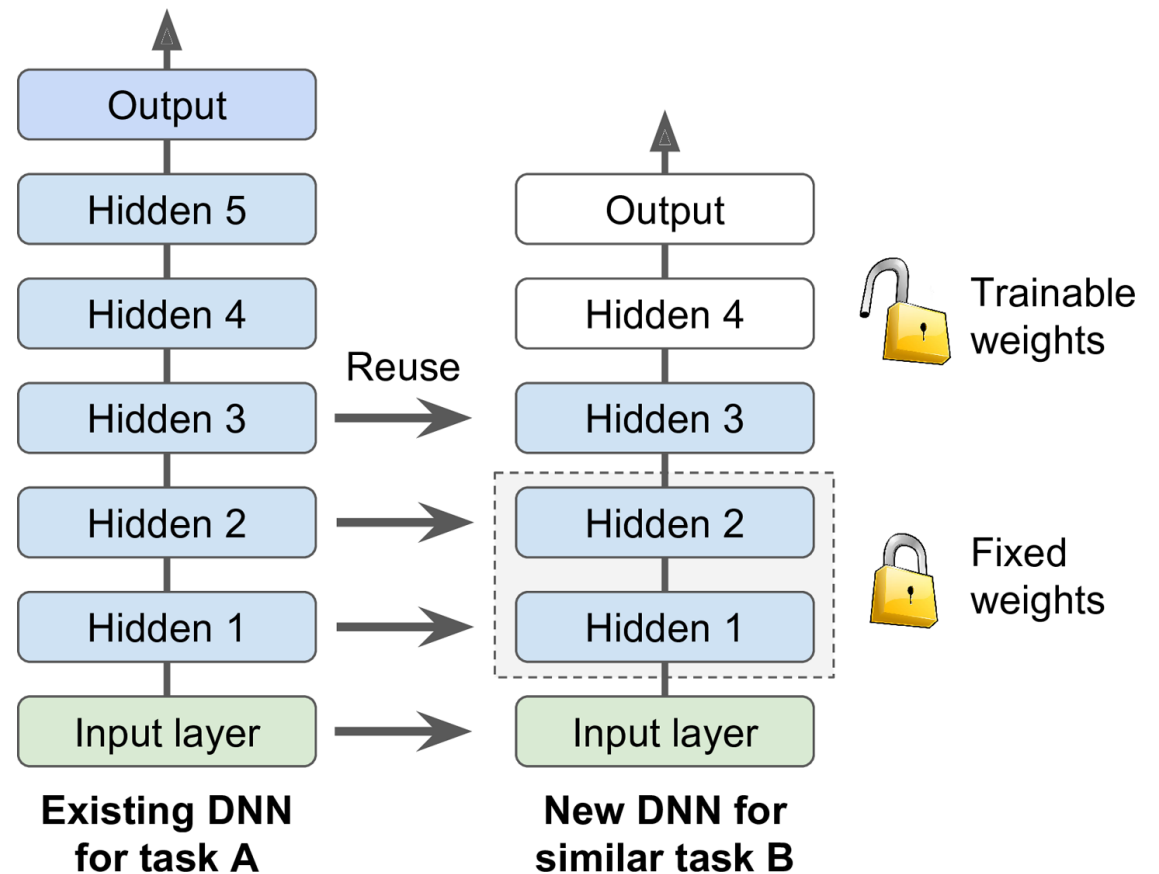
```
model.compile(loss="mse", optimizer=optimizer)
```

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

3. Reusing Pretrained Layers

- **Transfer Learning:** Using one NN developed for a certain task to solve another task.
- Useful to **shorten training time** or with **small datasets**.



Transfer Learning with Keras

```
# Load the ready model
model_A = keras.models.load_model("my_model_A.h5")
# Create a new model using all but the last layer
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
# Freeze loaded layers then compile
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy",
                    optimizer="sgd", metrics=["accuracy"])
```


Transfer Learning with Keras

```
# Train the model for a few epochs
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))
# Unfreeze loaded layers
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True
# Compile with small learning rate (default = 1e-2)
optimizer = keras.optimizers.SGD(lr=1e-4)
model_B_on_A.compile(loss="binary_crossentropy",
                    optimizer=optimizer, metrics=["accuracy"])
```

Transfer Learning with Keras

```
# Train the model for more epochs
```

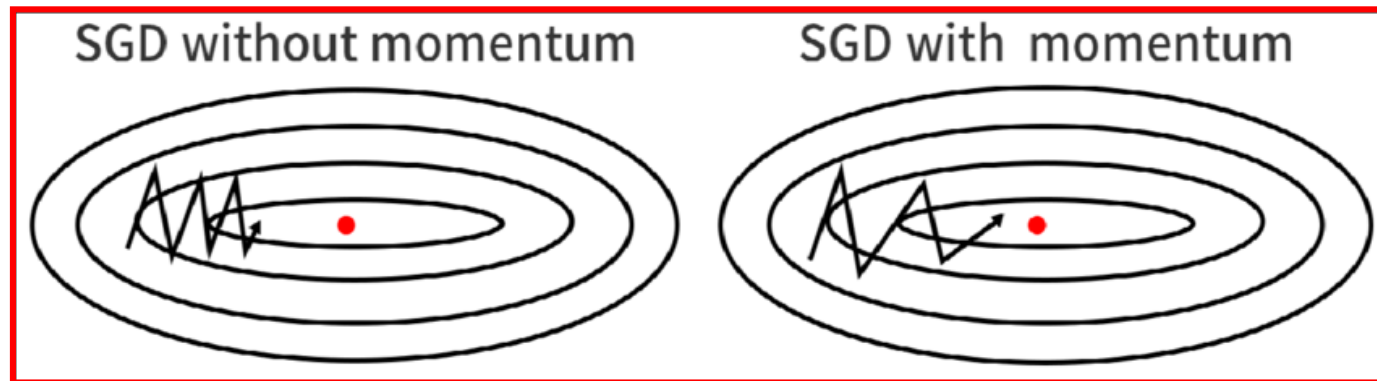
```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,  
                           validation_data=(X_valid_B, y_valid_B))
```

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

4. Faster Optimizers

- The SGD optimizer can be made faster using **momentum optimization**



$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$$

$$1. \quad \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$

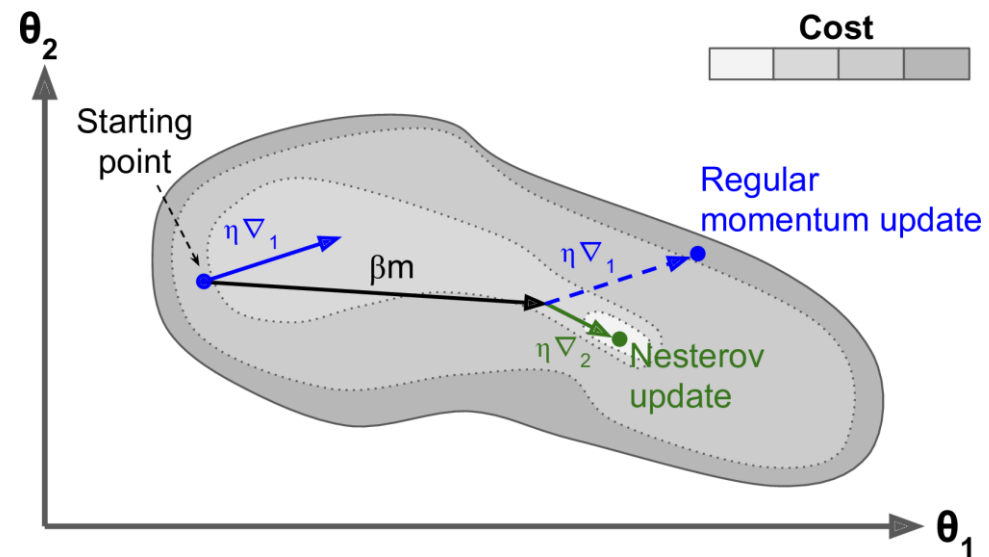
β

`optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)`

4. Faster Optimizers

- **Nesterov momentum optimization** measures the gradient of the cost function not at the local position θ but slightly ahead in the direction of the momentum, at $\theta + \beta\mathbf{m}$

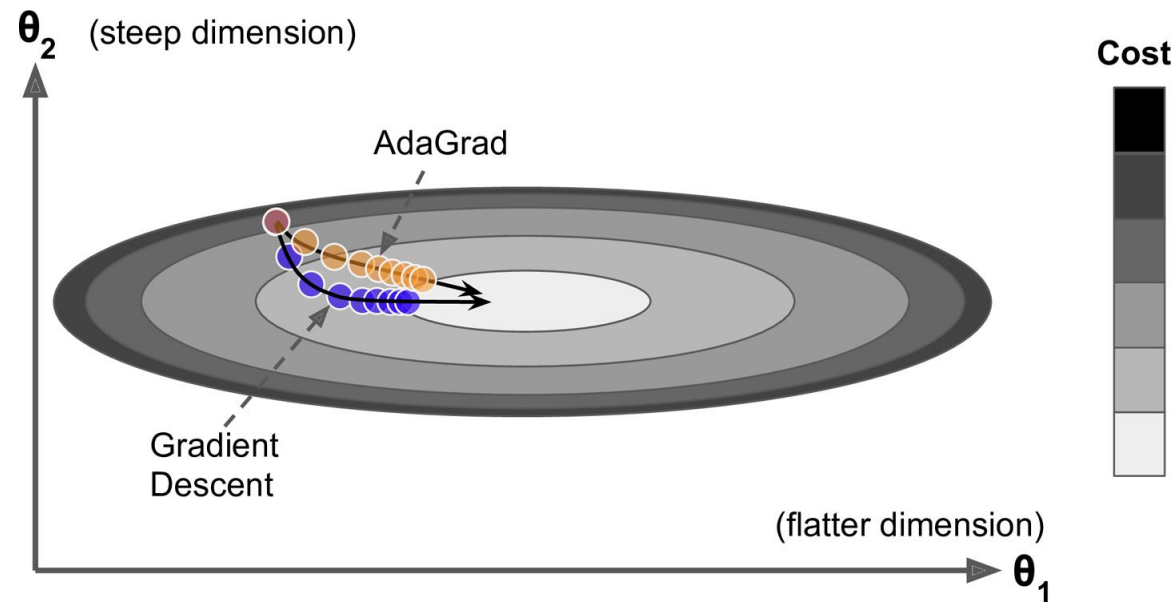
1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$



```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9,  
                                  nesterov=True)
```

4. Faster Optimizers

- The **adaptive optimizers** such as **AdaGrad**, **RMSProp**, **Adam**, and **Nadam** scale down the gradient vector along the steepest dimensions.



```
optimizer = keras.optimizers.RMSprop()  
optimizer = keras.optimizers.Adam()
```

4. Faster Optimizers

- RMSProp, Adam and Nadam often **converge fast**. But they can give poor **generalization**.
- Solution: Use Nesterov accelerated gradient.

Class	Speed	Quality
SGD	*	***
SGD with momentum, Nesterov	**	***
Adagrad	***	*
RMSProp, Adam, Nadam, AdaMax	***	** or ***

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

5. Avoiding Overfitting

- Deep neural networks typically have many parameters, giving them **ability to fit** a huge variety of complex datasets.
- **Useful regularization techniques:**
 - Early stopping
 - Batch normalization
 - ℓ_1 and ℓ_2 regularization
 - Dropout

5.1 ℓ_1 and ℓ_2 Regularization

- Constrain a neural network's connection weights.

- ℓ_1 :
$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m} * \sum \|w\|$$

- ℓ_2 :
$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m} * \sum \|w\|^2$$

```
layer = keras.layers.Dense(100, activation="elu",  
                             kernel_initializer="he_normal",  
                             kernel_regularizer=keras.regularizers.l1(0.01))
```

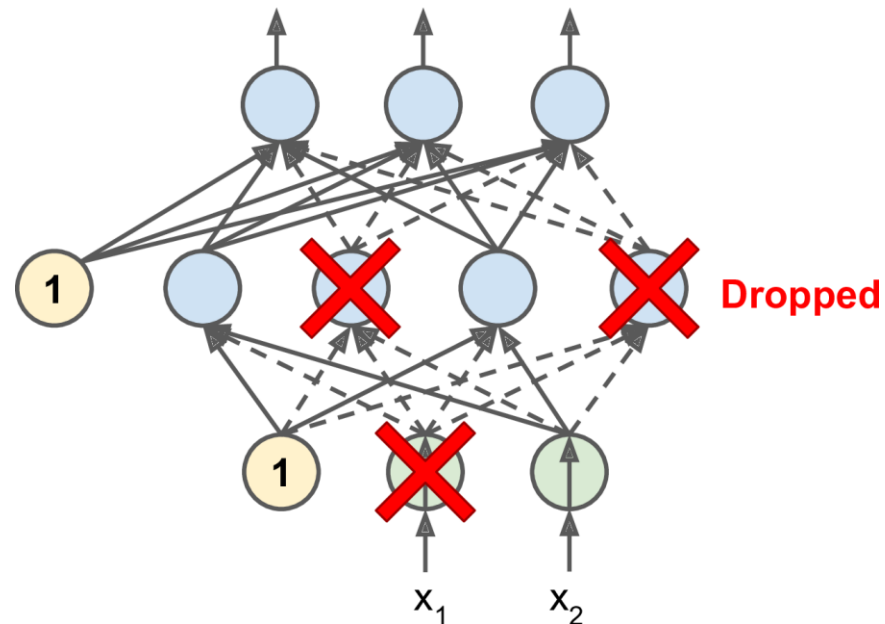
The other regularization functions:

```
keras.regularizers.l2(0.01)
```

```
keras.regularizers.l1_l2(l1=0.01, l2=0.01)
```

5.2 Dropout

- Popular technique to improve accuracy.
- At every training step, every neuron (excluding the output neurons) has a probability p of being temporarily **dropped out**.



5.2 Dropout

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu",
        kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu",
        kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

6. Summary

- **Recommended default DNN** configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1 cycle

6. Summary

- For a simple **stack of dense** or **CNN layers**.

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1 cycle

7. Exercise

11.8. Practice training a deep neural network on the **CIFAR10 image dataset**:

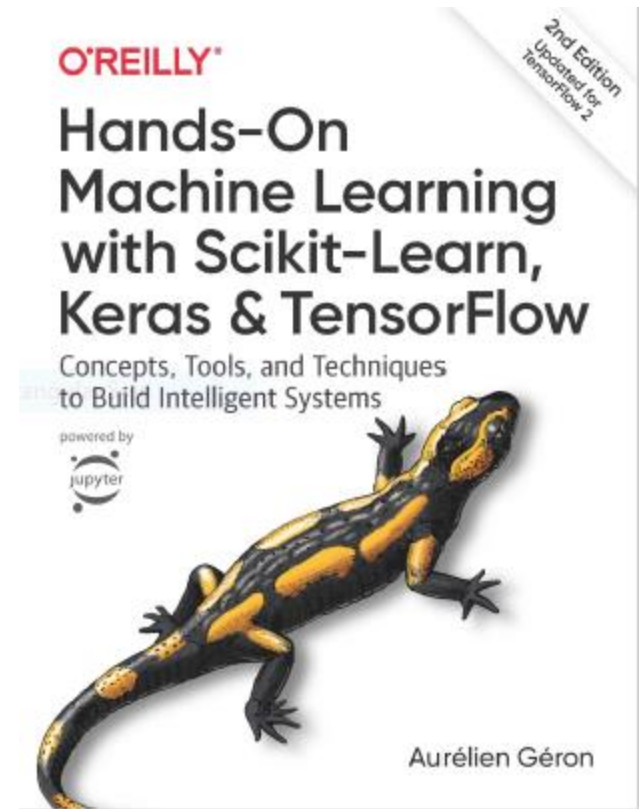
- a) Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the ELU activation function.
- b) Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with `keras.datasets.cifar10.load_data()`. The dataset is composed of 60,000 32 × 32-pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you'll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model's architecture or hyperparameters.
- c) Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?
- d) Try replacing Batch Normalization with SELU, and make the necessary adjustments to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
- e) Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.
- f) Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.

Deep Computer Vision Using Convolutional Neural Networks

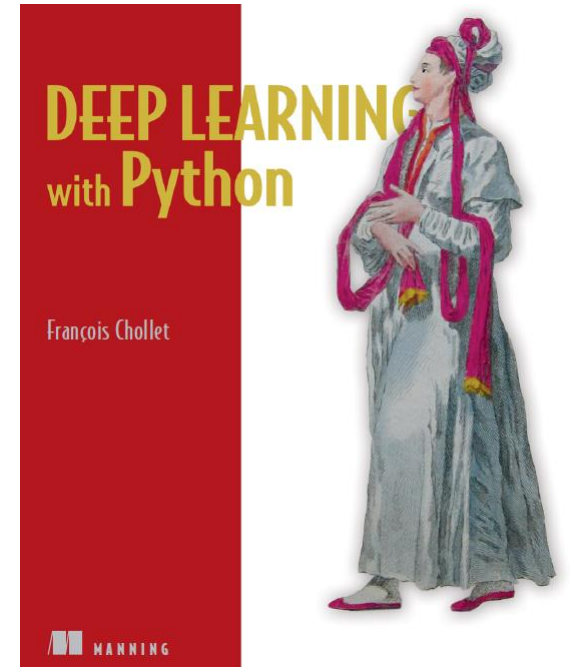
Prof. Gheith Abandah

Reference

- Chapter 14: **Deep Computer Vision Using Convolutional Neural Networks**
- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>



Reference



- **Deep Learning with Python**, by François Chollet, Manning Pub. 2018

Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

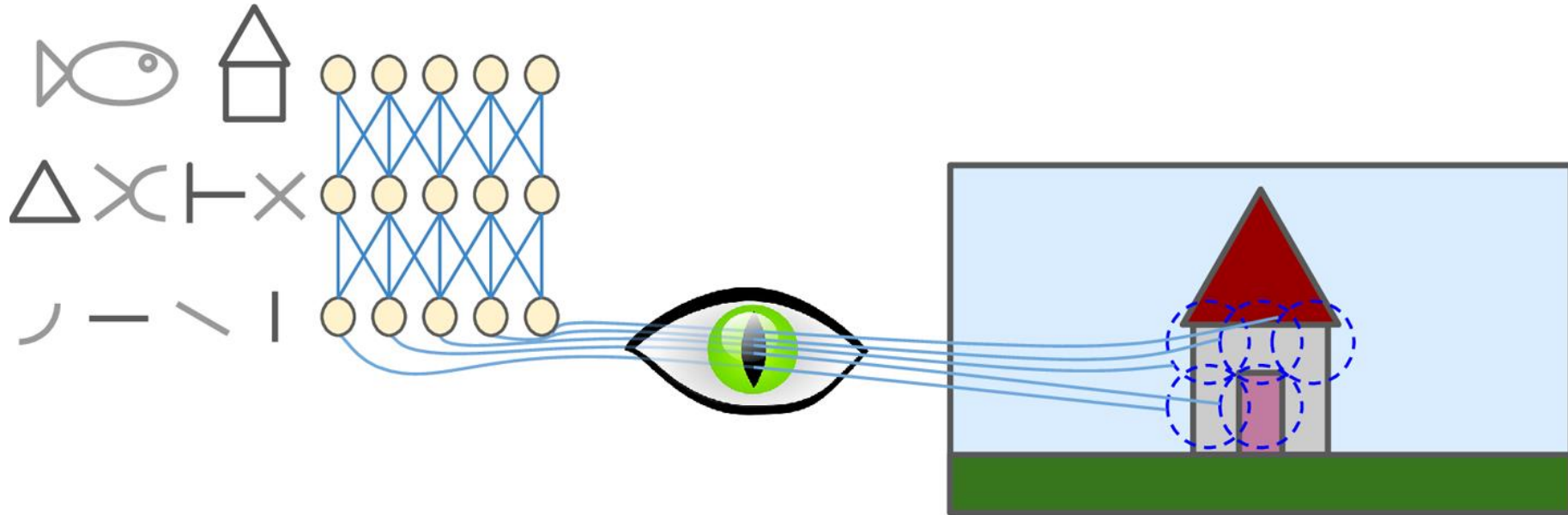
Introduction

- YouTube Video: **Convolutional Neural Networks (CNNs) explained** from Deeplizard

https://youtu.be/YRhxdVk_sIs

1. Introduction

- **Convolutional neural networks (CNNs)** emerged from the study of the brain's **visual cortex**.
- Many neurons in the visual cortex have a small **local receptive field**.

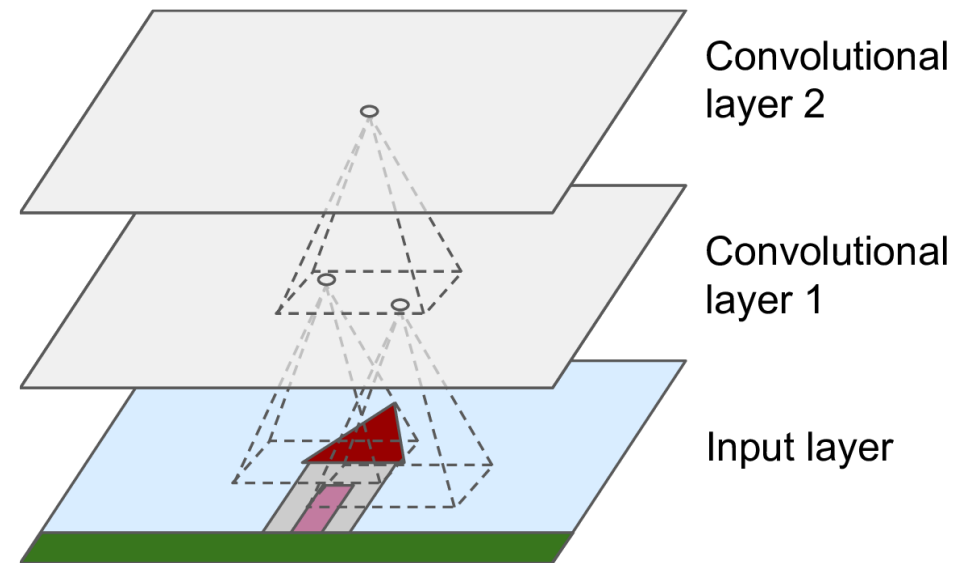


Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

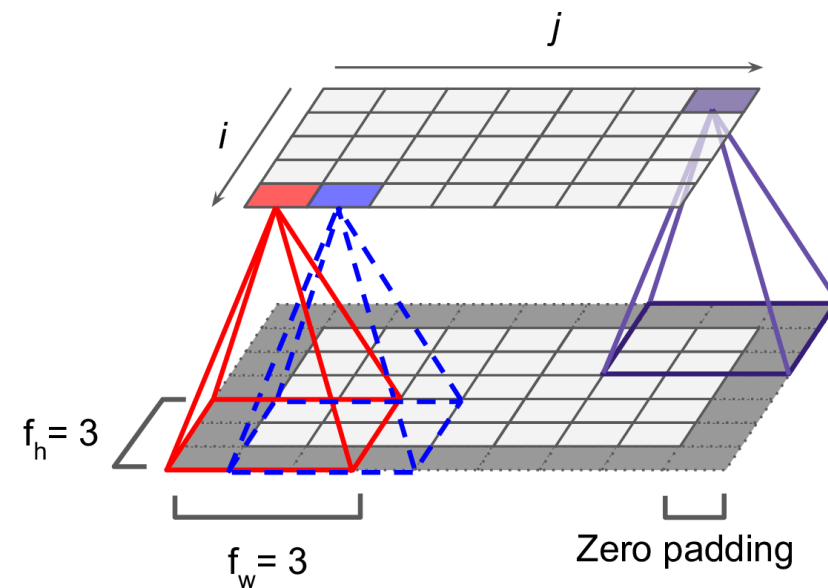
2. Convolutional Layer

- **Neurons** in one layer are not connected to every single pixel/neuron in the previous layer, but only to pixels/neurons in their **receptive fields**.
- This architecture allows the network to concentrate on **low-level features** in one layer, then assemble them into **higher-level features** in the next layer.
- Each layer is represented in **2D**.



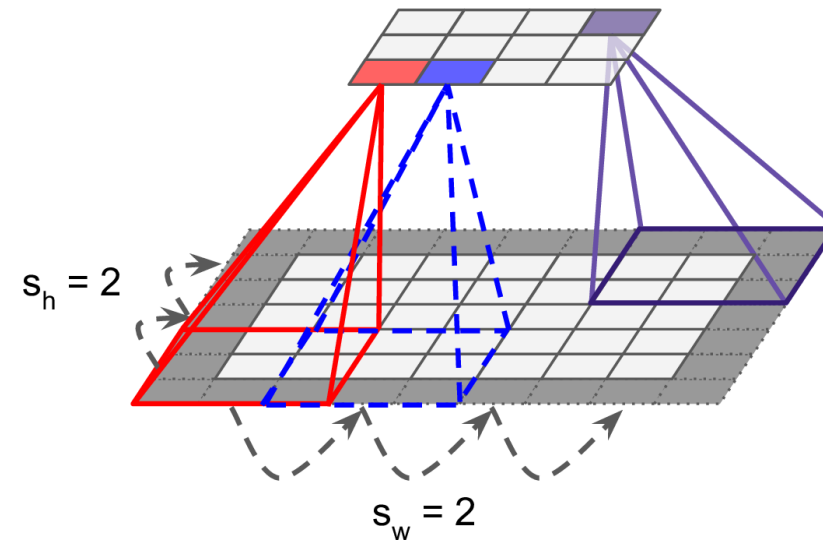
2. Convolutional Layer

- f_h and f_w are the height and width of the receptive field.
- **Zero padding**: In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs.



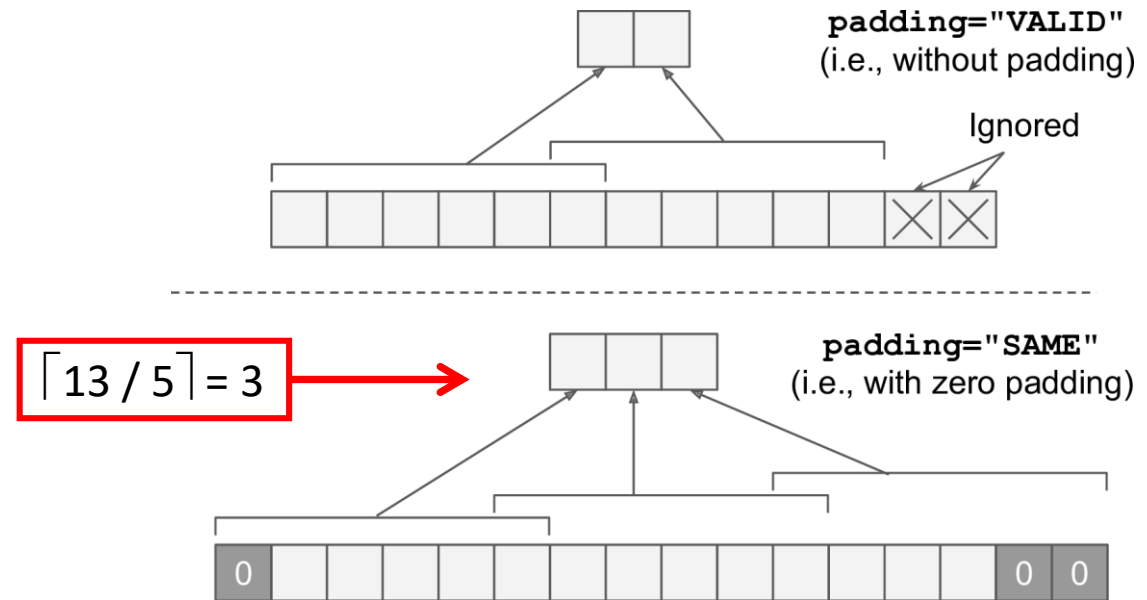
2. Convolutional Layer

- It is also possible to connect a large input layer to a smaller layer by **spacing out** the receptive fields.
- The distance between two consecutive receptive fields is called the **stride**.
- A neuron located in row i , column j is connected to the neurons in the previous layer located in:
 - Rows: $i \times s_h$ to $i \times s_h + f_h - 1$
 - Cols: $j \times s_w$ to $j \times s_w + f_w - 1$



2. Convolutional Layer

- Keras supports
 - **No padding** (default)
`padding="VALID"`
 - **Zero padding**
`padding="SAME"`
- Example:
 - Input width: 13
 - Filter width: 6
 - Stride: 5



2. Convolutional Layer

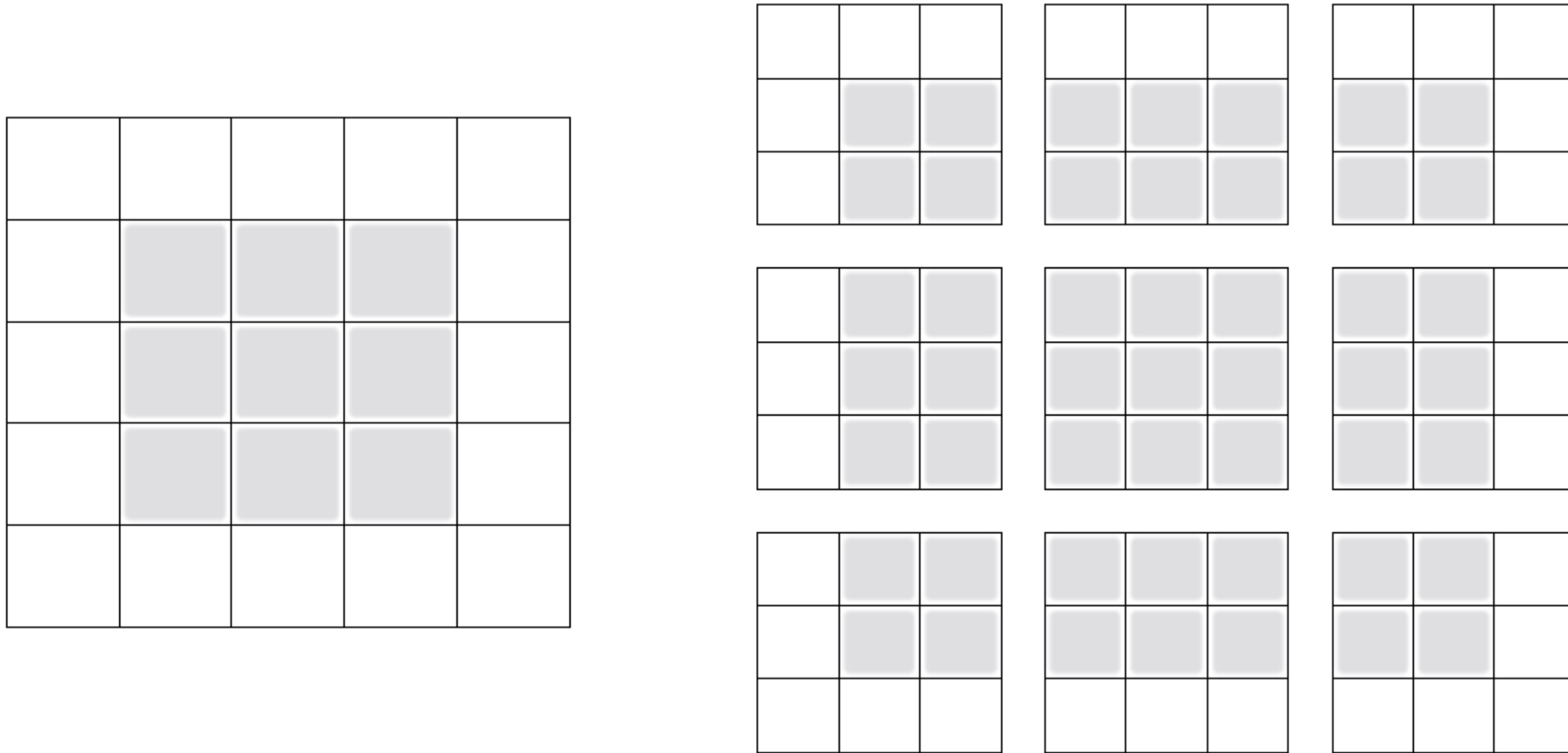
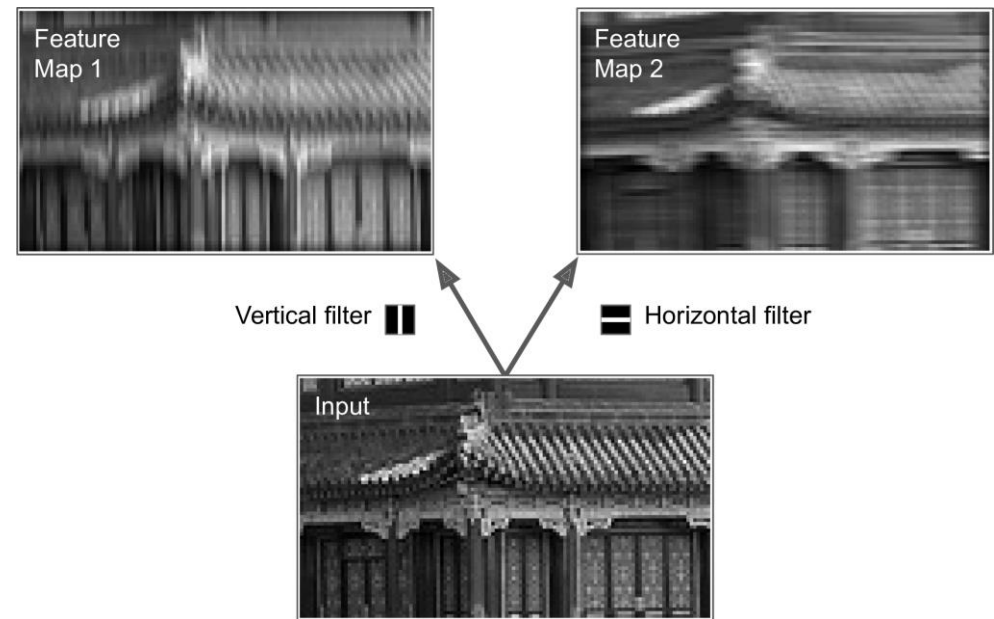


Figure 5.5 Valid locations of 3×3 patches in a 5×5 input feature map

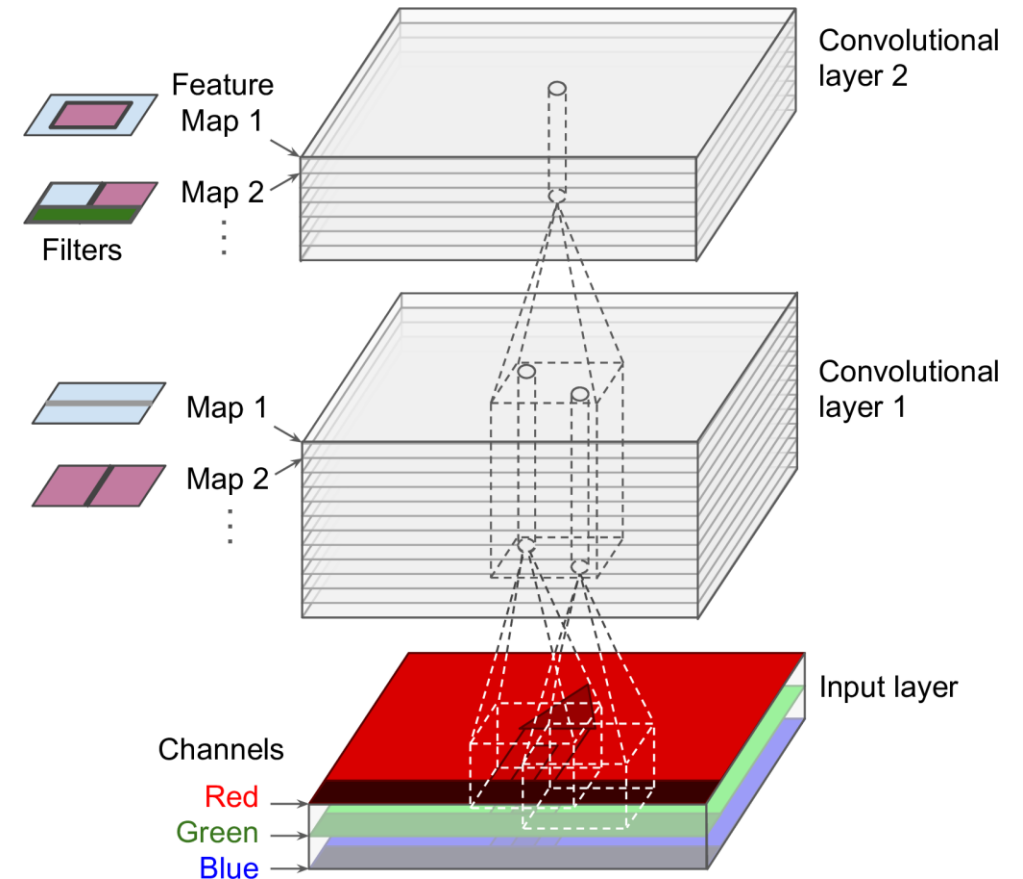
2.1 Filters

- A neuron's weights can be represented as a small image the size of the receptive field, called **filters**.
- When all neurons in a layer use the same line filters, we get the **feature maps** on the top.



2.2 Stacking Feature Maps

- In reality, each layer is **3D** composed of **several feature maps** of equal sizes.
- **Within** one feature map, all neurons **share** the same parameters, but **different** feature maps may have **different** parameters.
- Once the CNN has learned to **recognize a pattern in one location**, it can recognize it in any other location.



2.3 Mathematical Summary

Equation 14-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \quad \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ is the **output** of the neuron located in row i , column j in feature map k
- $f_{n'}$ is the number of **feature maps** in the previous layer

2.4 Memory Requirements

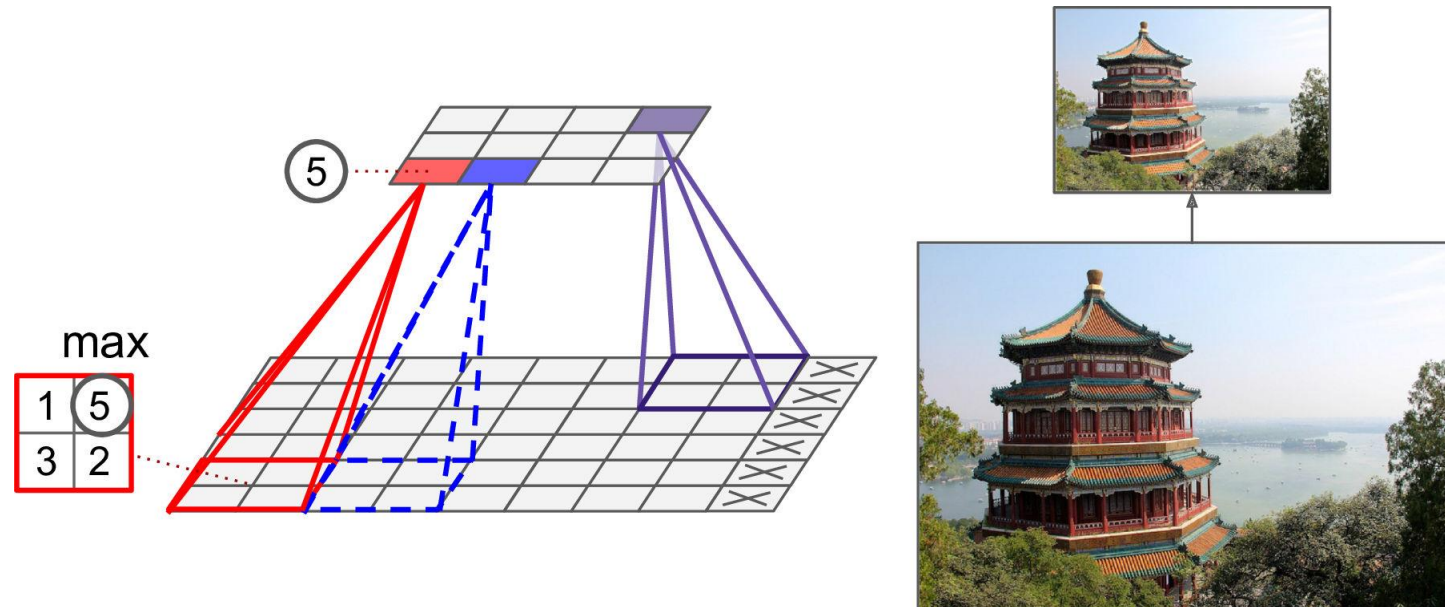
- Convolutional layers require a **huge amount of RAM**.
- **Example**: Convolutional layer with 5×5 filters, 200 feature maps of size 150×100 , with stride 1 and "same" padding. Input is RGB image (three channels).
 - Parameters = $(5 \times 5 \times 3 + 1) \times 200 = 15,200$
 - Size of feature maps (single precision) = $200 \times 150 \times 100 \times 4 = 12$ MB of RAM
 - 1.2 GB of RAM for a mini batch of 100 instances

Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

3. Pooling Layer

- Its goal is to **subsample** (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters.
- It aggregates the inputs using **max** or **mean**.

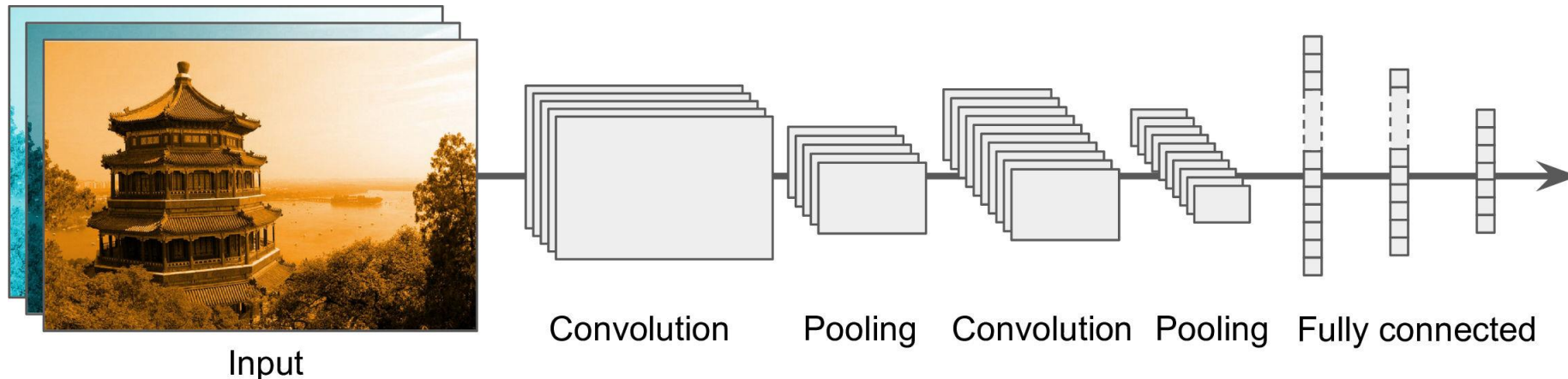


Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

4. CNN Architectures

- **Stack** few **convolutional layers** (each one generally followed by a **ReLU** layer), then a **pooling** layer, then another few convolutional layers, then another pooling layer, and so on. The image gets **smaller and smaller**, but it also gets **deeper and deeper**. At the end, a **dense NN** is added.



4.1 Example – Fashion MNIST

```
model = keras.models.Sequential([  
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",  
        input_shape=[28, 28, 1]),  
    keras.layers.MaxPooling2D(2),  
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),  
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),  
    keras.layers.MaxPooling2D(2),  
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),  
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),  
    keras.layers.MaxPooling2D(2),  
    keras.layers.Flatten(),  
    keras.layers.Dense(128, activation="relu"),  
    keras.layers.Dropout(0.5),  
    keras.layers.Dense(64, activation="relu"),  
    keras.layers.Dropout(0.5),  
    keras.layers.Dense(10, activation="softmax")  
])
```

Filter size

Feature maps

2x2 window and stride 2

1)

4.1 Example – Fashion MNIST

```
model.compile(loss="sparse_categorical_crossentropy",  
              optimizer="nadam", metrics=["accuracy"])
```

```
history = model.fit(X_train, y_train, epochs=10,  
                   validation_data=(X_valid, y_valid))
```

Train on 55000 samples, validate on 5000 samples

```
Epoch 1/10 55000/55000 [=====] - 51s 923us/sample - loss:  
0.7183 - accuracy: 0.7529 - val_loss: 0.4029 - val_accuracy: 0.8510
```

...

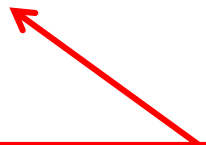
```
Epoch 10/10
```

```
55000/55000 [=====] - 50s 911us/sample - loss: 0.2561 -  
accuracy: 0.9145 - val_loss: 0.2891 - val_accuracy: 0.9036
```

4.1 Example – Fashion MNIST

```
score = model.evaluate(X_test, y_test)
X_new = X_test[:10] # pretend we have new images
y_pred = model.predict(X_new)
```

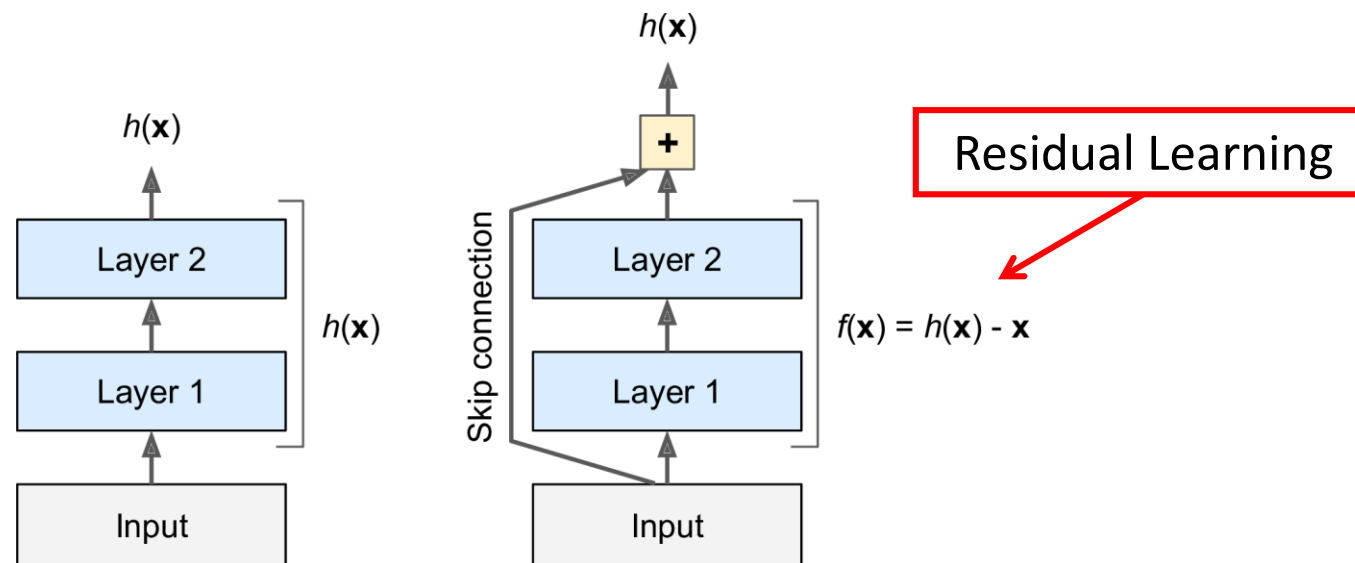
```
10000/10000 [=====] - 2s 239us/sample - loss:
0.2972 - accuracy: 0.8983
```



Can reach 92% with
more epochs

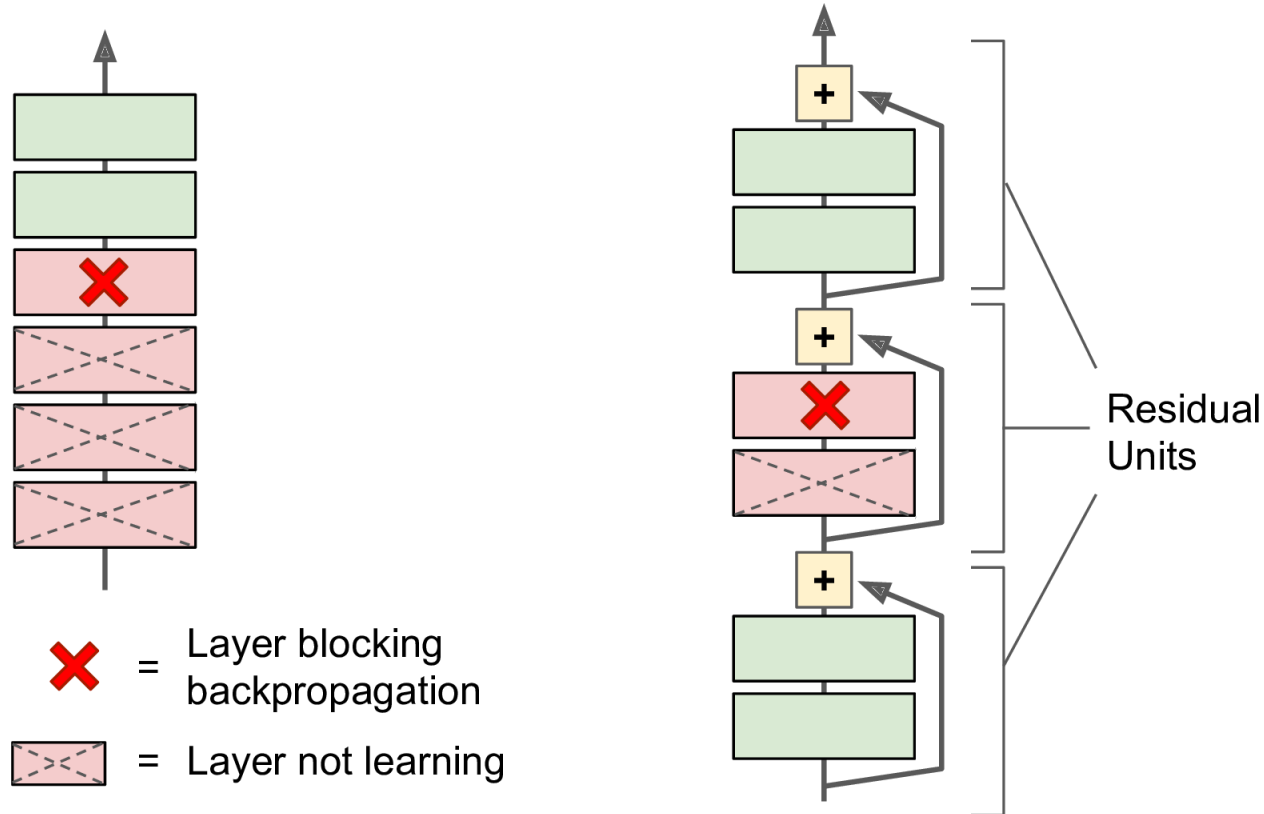
4.2 ResNet

- **Residual Network** (or ResNet) won the ILSVRC 2015 challenge.
- Top-5 error rate under 3.6%, using an extremely deep CNN composed of **152 layers**.
- To train such a deep network, it uses **skip connections**.



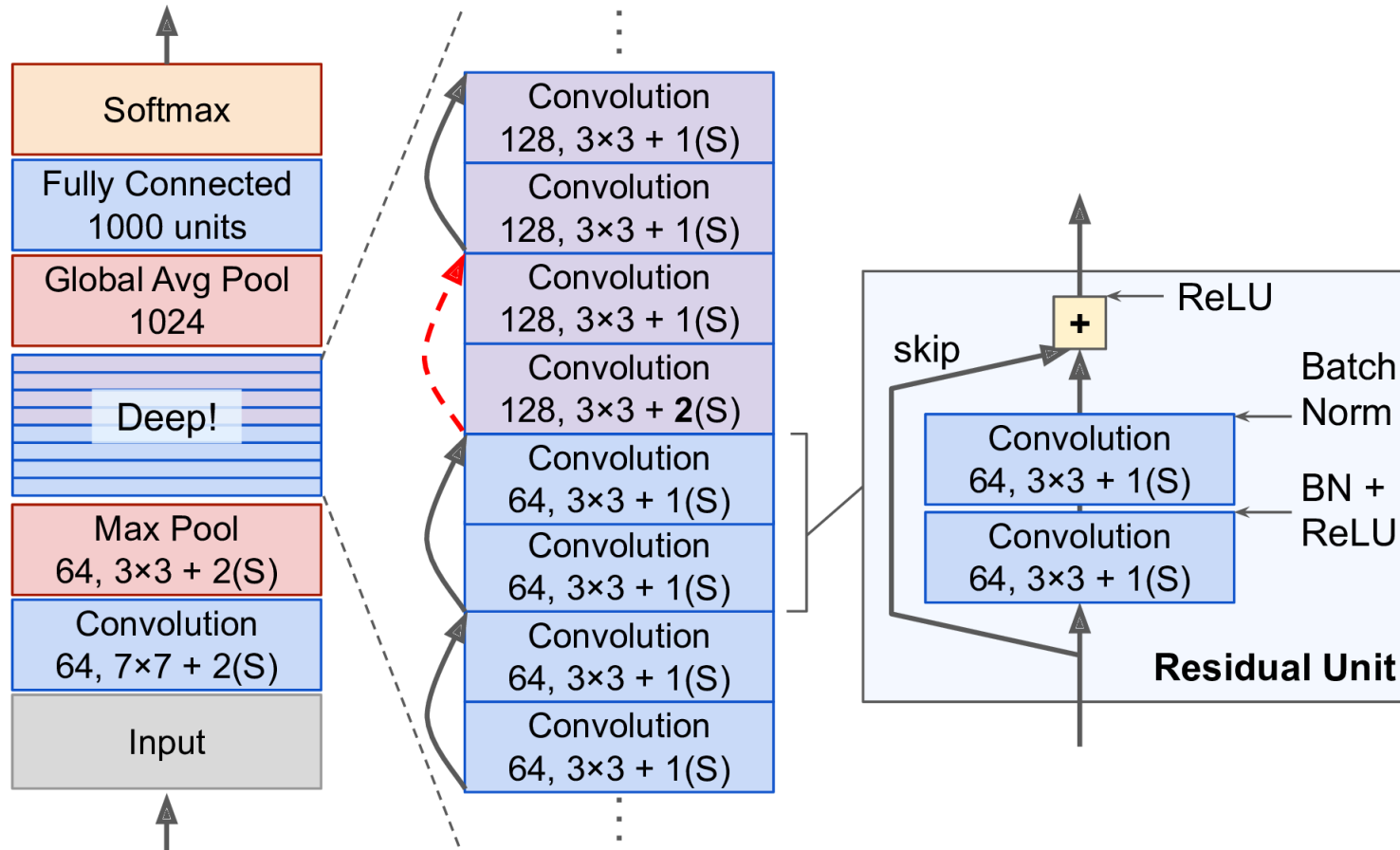
4.2 ResNet

- The network can start making progress even if several layers have not started learning yet.



4.2 ResNet

- ResNet is a **stack** of residual units.



Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

5. Using Pretrained Models

- Pretrained networks are readily available from the **keras.applications** package.
- Check <https://github.com/keras-team/keras-applications>
- You can load the **ResNet-50** model, pretrained on **ImageNet**, with the following line of code:

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

5. Using Pretrained Models

```
# Input: 224 × 224-pixel images
```

```
images_resized = tf.image.resize(images, [224, 224])
```

```
# Preprocess images, should be scaled 0-255
```

```
inputs = keras.applications.resnet50.preprocess_input(  
    images_resized * 255)
```

```
Y_proba = model.predict(inputs)
```

```
# Get top predictions out of the 1000-class probs.
```

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
```

5. Using Pretrained Models

```
# Print results
for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print(" {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

Image #0

n03877845	- palace	42.87%
n02825657	- bell_cote	40.57%
n03781244	- monastery	14.56%

Image #1

n04522168	- vase	46.83%
n07930864	- cup	7.78%
n11939491	- daisy	4.87%

Correct Class



Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

6. Pretrained Models for Transfer Learning

- Training a pretrained network (**Xception**) for a dataset from TFDS (<https://www.tensorflow.org/datasets>).
- **tf_flowers**: 3670 images, 5 classes

Load the dataset

```
import tensorflow_datasets as tfds
```

```
dataset, info = tfds.load("tf_flowers",  
                          as_supervised=True, with_info=True)
```

```
dataset_size = info.splits["train"].num_examples # 3670
```

```
n_classes = info.features["label"].num_classes # 5
```

```
class_names = info.features["label"].names
```



6. Pretrained Models for Transfer Learning

```
# Reload the dataset with three splits tf.data.Dataset
test_set_raw, valid_set_raw, train_set_raw = tfds.load(
    "tf_flowers", split=["train[:10%]",
                        "train[10%:25%]", "train[25%:]"],
    as_supervised=True)
```

```
# Define the preprocessing function
```

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(
        resized_image)
    return final_image, label
```

6. Pretrained Models for Transfer Learning

```
# Apply this preprocessing function to the 3 datasets
# Shuffle the training set
# Add batching and prefetching to all the datasets

batch_size = 32

train_set = train_set_raw.shuffle(3000).repeat()
train_set = train_set.map(preprocess).batch(
    batch_size).prefetch(1)

valid_set = valid_set_raw.map(preprocess).batch(
    batch_size).prefetch(1)

test_set = test_set_raw.map(preprocess).batch(
    batch_size).prefetch(1)
```

6. Pretrained Models for Transfer Learning

```
# Load an Xception model, pretrained on ImageNet
# excluding the global avg pool. and dense o/p layers
base_model = keras.applications.xception.Xception(
    weights="imagenet", include_top=False)

# Add global avg pool. layer based on model output
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, # Add dense o/p
    activation="softmax")(avg)
model = keras.models.Model(inputs=base_model.input,
    outputs=output) # Create the Keras Model
```

6. Pretrained Models for Transfer Learning

```
# Freeze the weights of the pretrained layers
```

```
for layer in base_model.layers:
```

```
    layer.trainable = False
```

```
# Compile the model and start training
```

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9,  
    decay=0.01) # LR=0.2 with scheudle, k=1/0.01
```

```
model.compile(loss="sparse_categorical_crossentropy",  
    optimizer=optimizer, metrics=["accuracy"])
```

```
history = model.fit(train_set, epochs=5,  
    validation_data=valid_set) # Tops at 75-80% acc.
```

$$\eta(t) = \eta_0 / (1 + t/k)$$



6. Pretrained Models for Transfer Learning

```
# Unfreeze the weights of the pretrained layers
for layer in base_model.layers:
    layer.trainable = True

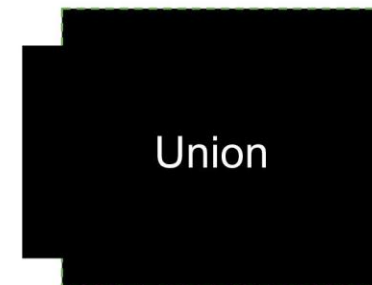
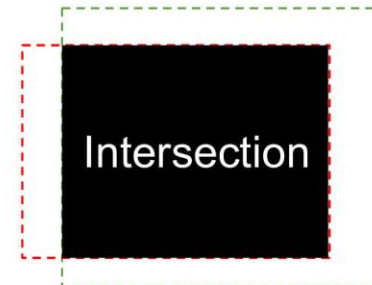
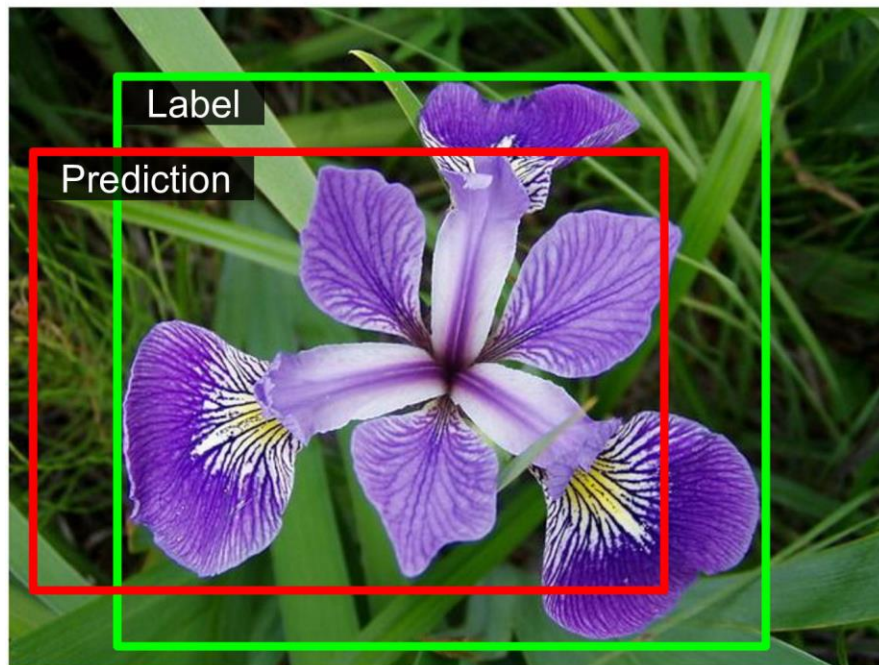
# Recompile with lower LR and decay
optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9,
    nesterov=True, decay=0.001)
model.compile(loss="sparse_categorical_crossentropy",
    optimizer=optimizer, metrics=["accuracy"])
history = model.fit(train_set, epochs=40,
    validation_data=valid_set) # Result: 95% acc.
```

Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
- 7. Classification and localization**
- 8. Object detection**
- 9. Semantic segmentation**
- 10. Exercises**

7. Classification and Localization

- **Localizing** an object in a picture can be expressed as a **regression** task.
- Predict the horizontal and vertical coordinates of the object's center and its height and width.



Common metric:
the Intersection
over Union (IoU)

7. Classification and Localization

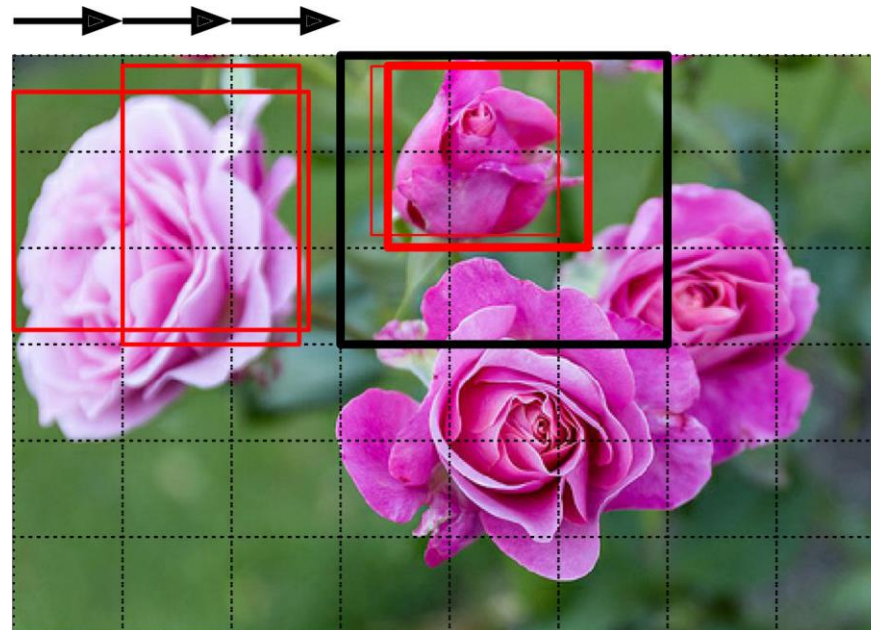
```
base_model = keras.applications.xception.Xception(  
    weights="imagenet", include_top=False)  
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)  
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)  
loc_output = keras.layers.Dense(4)(avg)  
model = keras.Model(inputs=base_model.input,  
    outputs=class_output, loc_output)  
  
model.compile(loss=["sparse_categorical_crossentropy", "mse"],  
    loss_weights=[0.8, 0.2],  
    optimizer=optimizer, metrics=["accuracy"])
```


Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
- 8. Object detection**
- 9. Semantic segmentation**
10. Exercises

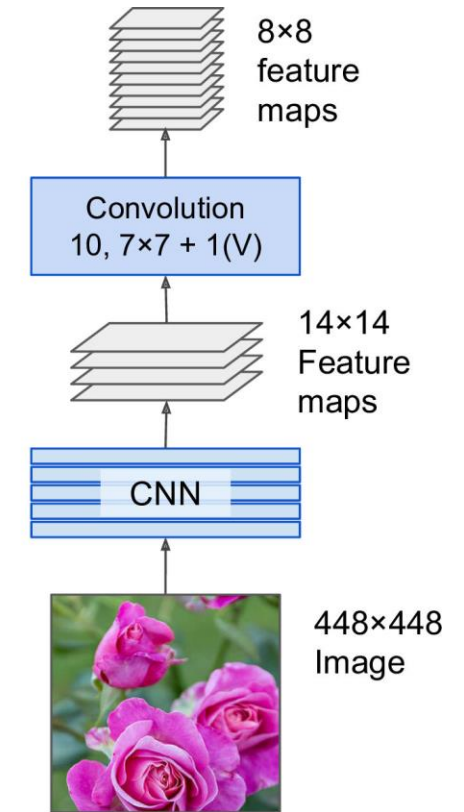
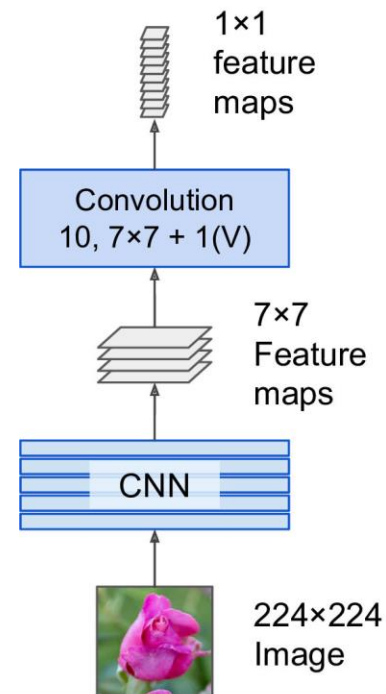
8. Object detection

- The task of **classifying and localizing** multiple objects in an image.
- A **slow** approach is use a CNN trained to classify and locate a single object, then **slide** it across the image.



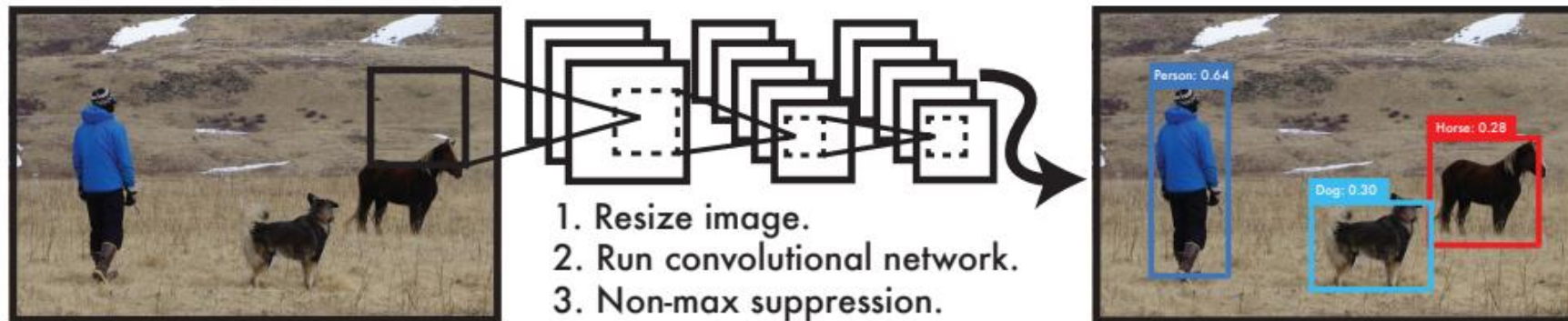
8.1 Fully Convolutional Networks

- FCN has also a **convolution layer at the output** with **valid padding**.
- FCN can process images of **any size**.
- **Example:**
 - Train the CNN for classification and localization on small images, 10 outputs.
 - For larger image, it output 8×8 grid where each cell contains 10 numbers.



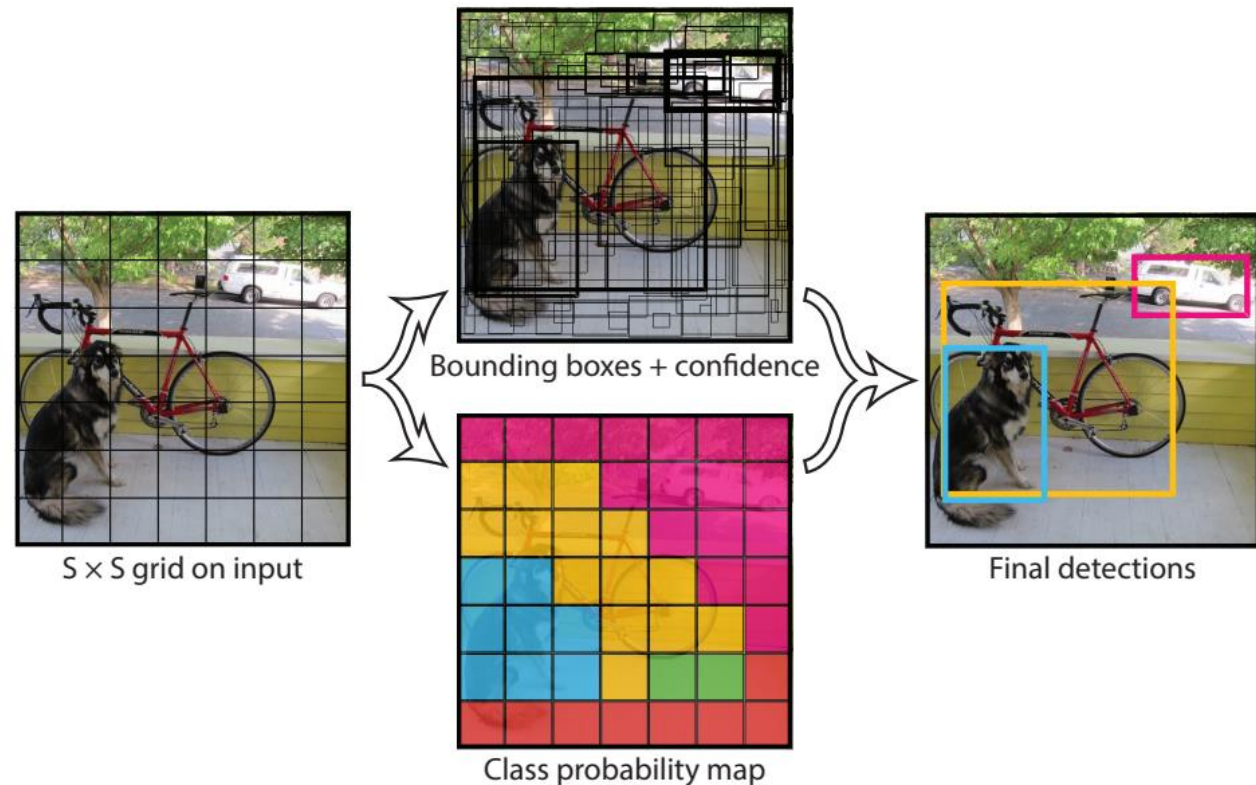
8.2 You Only Look Once (YOLO)

- **YOLO** is an extremely fast and accurate object detection architecture.
 1. Resizes the input image to 448×448
 2. Runs a single convolutional network on the image
 3. Thresholds the resulting detections by the model's confidence.



8.2 You Only Look Once (YOLO)

- Models detection as a regression problem. It divides the image into an $S \times S$ grid.
- For each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities.



Outline

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

9. Semantic Segmentation

- Each pixel is classified according to the class of the object it belongs to.
- Can use **FCN** followed by up **sampling** layers.



Exercises

- 14.9. Build your own CNN from scratch and try to achieve the highest possible accuracy on **MNIST**.
- 14.10. Use **transfer learning** for large image classification, going through these steps:
- Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can use an existing dataset (e.g., from TensorFlow Datasets).
 - Split it into a training set, a validation set, and a test set.
 - Build the input pipeline, including the appropriate preprocessing operations, and optionally add data augmentation.
 - Fine-tune a pretrained model on this dataset.

Summary

1. Introduction
2. Convolutional layer
 1. Filters
 2. Stacking feature maps
 3. Mathematical summary
 4. Memory requirements
3. Pooling layer
4. CNN architectures
 1. Example – Fashion MNIST
 2. ResNet
5. Using pretrained models
6. Pretrained models for transfer learning
7. Classification and localization
8. Object detection
9. Semantic segmentation
10. Exercises

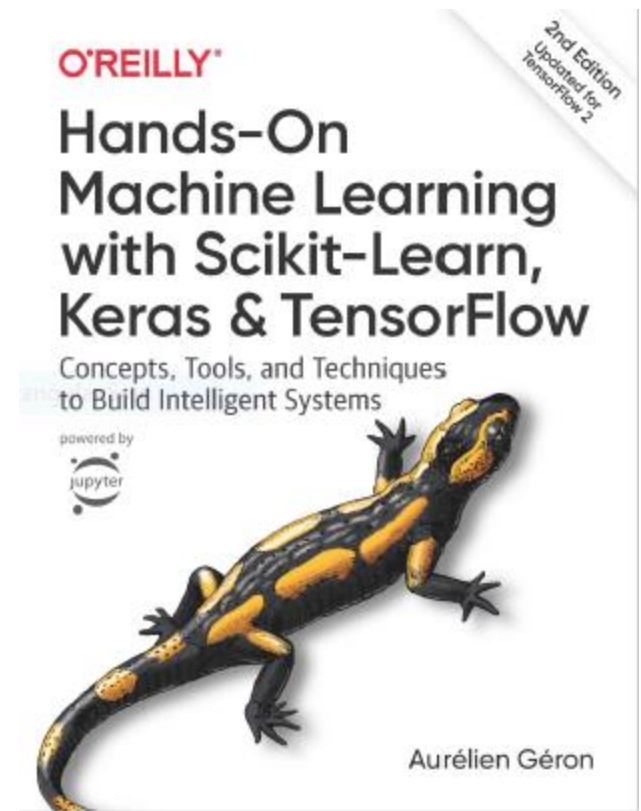
Recurrent Neural Networks

Prof. Gheith Abandah

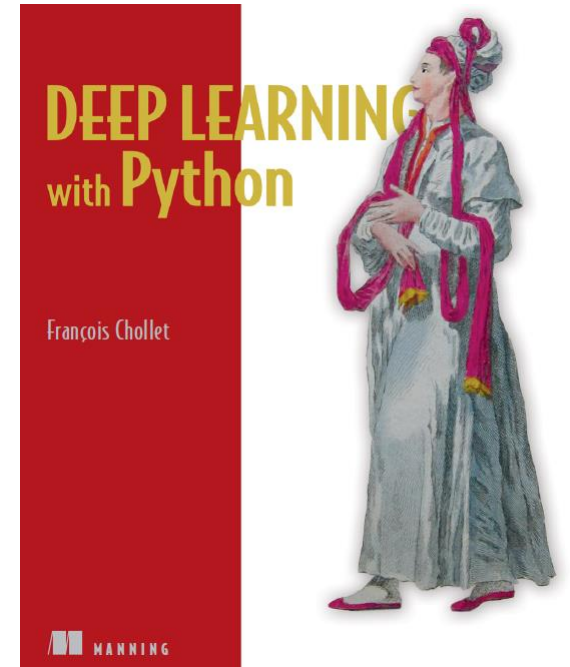
Reference

- Chapter 15: **Processing Sequences Using RNNs and CNNs**

- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>



Reference



- **Deep Learning with Python**, by François Chollet, Manning Pub. 2018

Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
 1. Implementing a simple RNN
 2. Deep RNNs
 3. Forecasting Several Time Steps Ahead
5. Handling long sequences
 1. LSTM cell
 2. GRU cell
6. Exercises

Introduction

- YouTube Video: **Deep Learning with Tensorflow - The Recurrent Neural Network Model** from Cognitive Class

<https://youtu.be/C0xoB8L8ms0>

1. Introduction

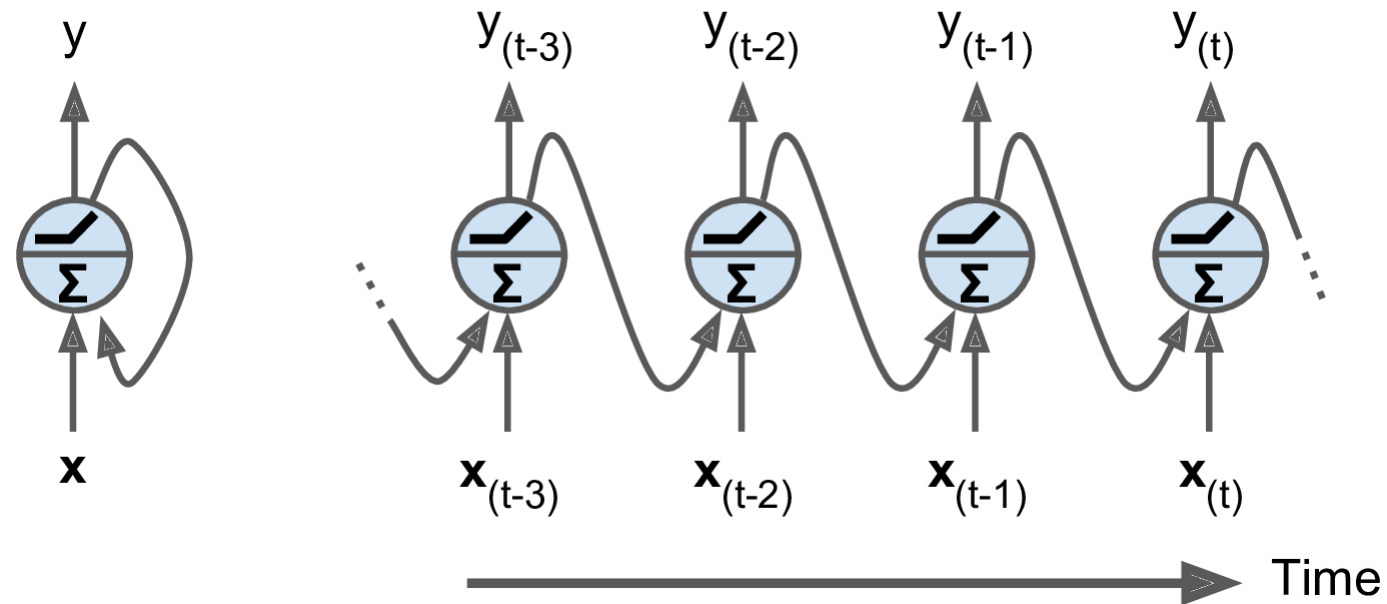
- **Recurrent neural networks (RNNs)** are used to handle time series data or sequences.
- **Applications:**
 - Predicting the future (stock prices)
 - Autonomous driving systems (predicting trajectories)
 - Natural language processing (automatic translation, speech-to-text, or sentiment analysis)
 - Creativity (music composition, handwriting, drawing)
 - Image analysis (image captions)

Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
 1. Implementing a simple RNN
 2. Deep RNNs
 3. Forecasting Several Time Steps Ahead
5. Handling long sequences
 1. LSTM cell
 2. GRU cell
6. Exercises

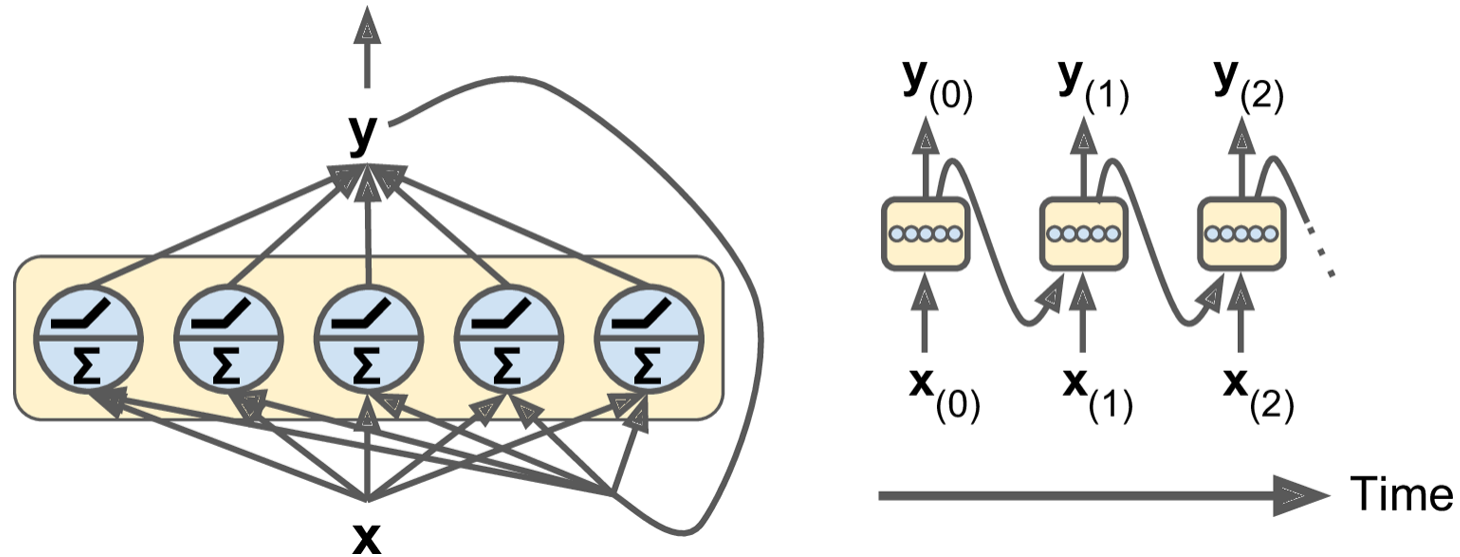
2. Recurrent Neurons and Layers

- The figure below shows a **recurrent neuron** (left), unrolled through time (right).



2. Recurrent Neurons and Layers

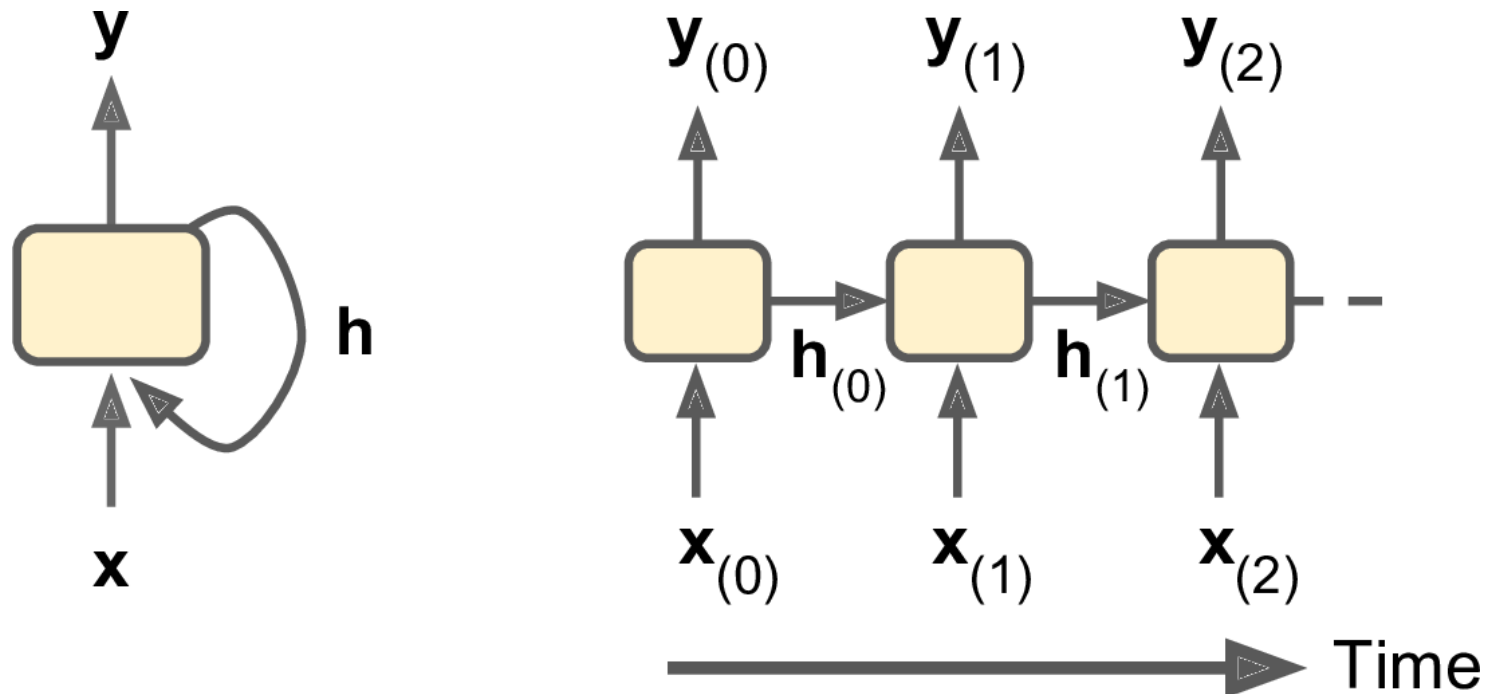
- Multiple recurrent neurons can be used in a **layer**.



- The **output** of the layer is:
$$Y_{(t)} = \phi(X_{(t)} \cdot W_x + Y_{(t-1)} \cdot W_y + b)$$

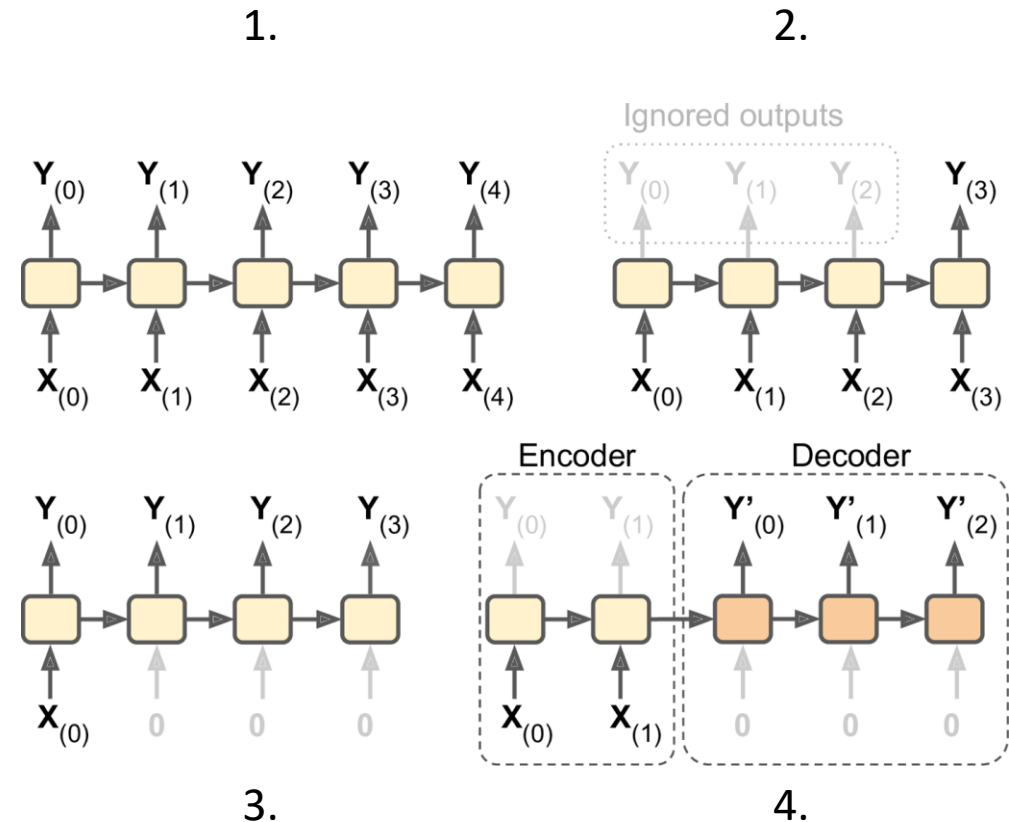
2. Recurrent Neurons and Layers

- Recurrent neurons have memory (hold state) and are called **memory cells**.
- The state $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$, not always $\equiv \mathbf{y}_{(t)}$



2. Recurrent Neurons and Layers: Input and Output Sequences

1. **Seq to seq net.:** For predicting the future.
2. **Seq to vector:** For analysis, e.g., sentiment score.
3. **Vector to seq:** For image captioning.
4. **Encoder-decoder:** For sequence transcription.

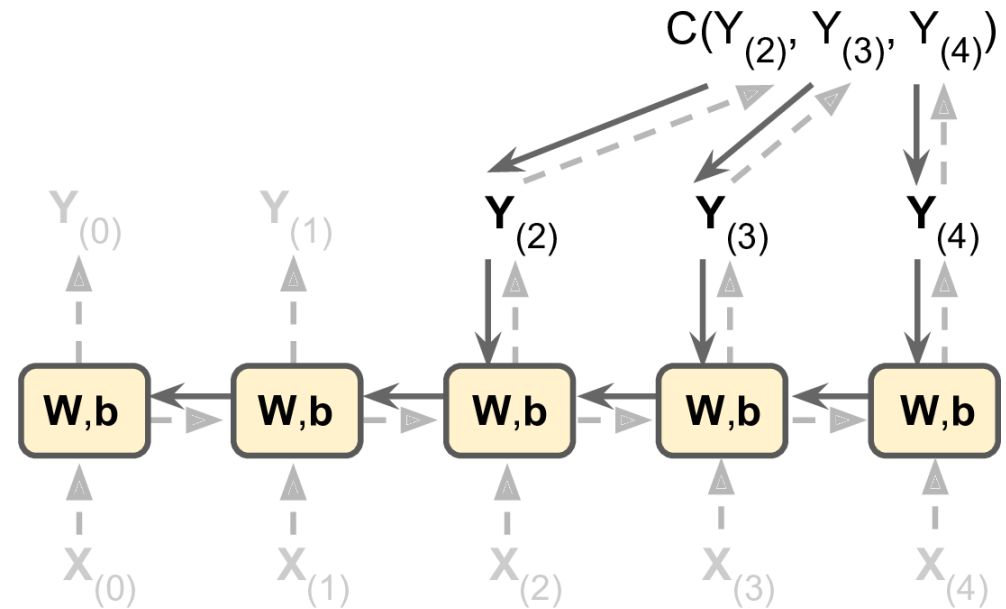


Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
 1. Implementing a simple RNN
 2. Deep RNNs
 3. Forecasting Several Time Steps Ahead
5. Handling long sequences
 1. LSTM cell
 2. GRU cell
6. Exercises

3. Training RNNs

- Training using strategy called **backpropagation through time** (BPTT).
- **Forward pass** (dashed)
- **Cost function** of the not-ignored outputs.
- **Cost gradients** are **propagated backward** through the unrolled network.

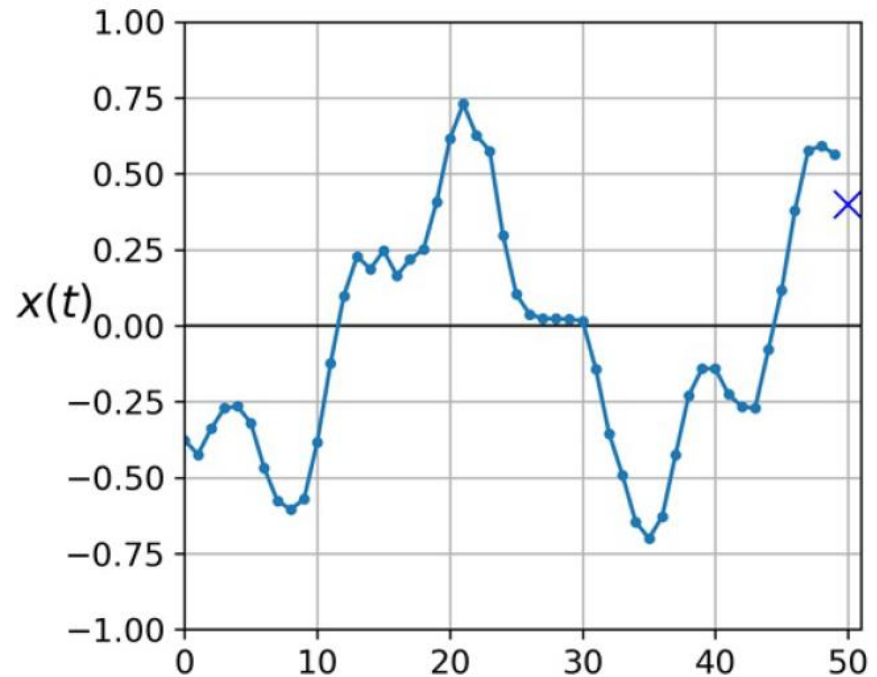


Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
 1. Implementing a simple RNN
 2. Deep RNNs
 3. Forecasting Several Time Steps Ahead
5. Handling long sequences
 1. LSTM cell
 2. GRU cell
6. Exercises

4. Forecasting a Time Series

- The data is a sequence of one or more values per **time step**.
 - **Univariate** time series
 - **Multivariate** time series
- **Forecasting**: predicting future values
 - Forecast the **next** value
 - Forecast **N next** values



4.1 Implementing a Simple RNN

```
# Generate 10,000 time series
n_steps = 50
series = generate_time_series(10000, n_steps + 1)

# Split them 7,000 : 2,000 : 1,000
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
                    # (7000, 50, 1), (7000, 1)
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]

X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

4.1 Implementing a Simple RNN

```
# Sequential model of one neuron
```

```
model = keras.models.Sequential([  
    keras.layers.SimpleRNN(1, input_shape=[None, 1])  
])
```

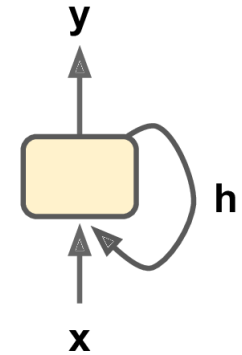
Uses tanh
activation $h_t = y_t$

```
optimizer = keras.optimizers.Adam(lr=0.005)
```

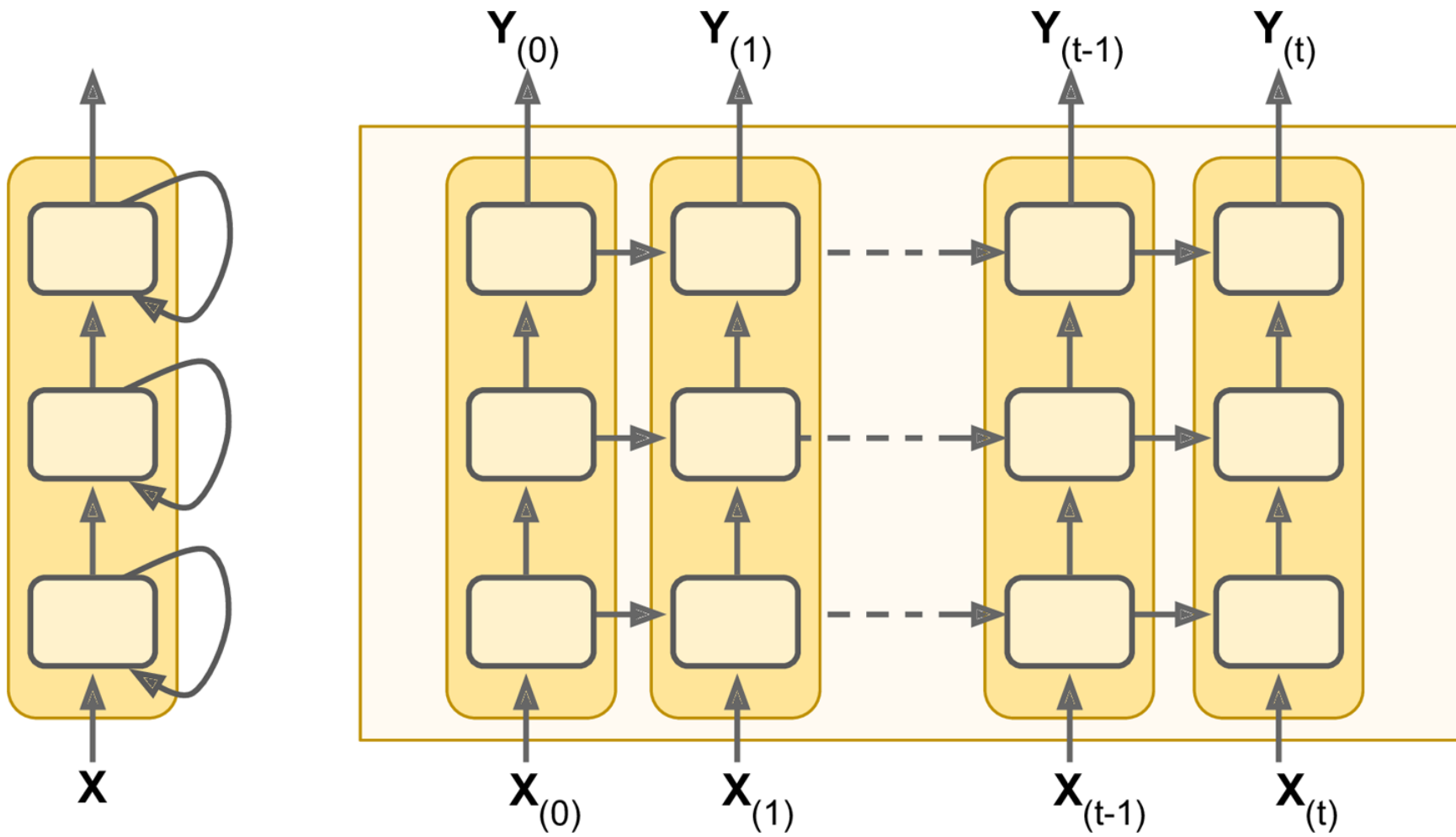
```
model.compile(loss="mse", optimizer=optimizer)
```

```
history = model.fit(X_train, y_train, epochs=20,  
                    validation_data=(X_valid, y_valid))
```

```
model.evaluate(X_valid, y_valid) # MSE = 0.011, Dense achieves 0.004
```



4.2 Deep RNNs



4.2 Deep RNNs

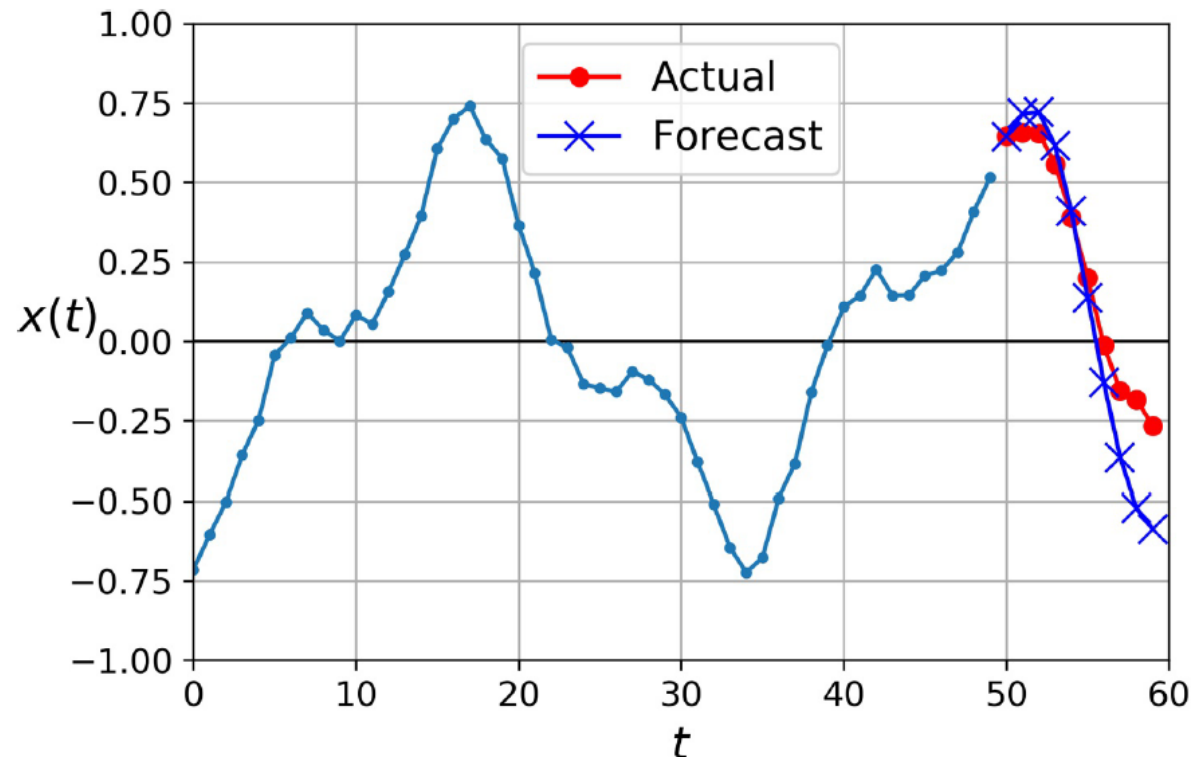
```
# Sequential model of two hidden RNN layers
```

```
model = keras.models.Sequential([  
    keras.layers.SimpleRNN(20,  
        return_sequences=True, # output all steps  
        input_shape=[None, 1]),  
    keras.layers.SimpleRNN(20),  
    keras.layers.Dense(1)  
])
```

```
# MSE = 0.0026
```

4.3 Forecasting Several Time Steps Ahead

- Can train an RNN to predict all **N next** values at once (sequence-to-vector model).
- The output layer should have N neurons.



4.3 Forecasting Several Time Steps Ahead

```
# Generate 10,000 time series with 10 steps ahead
series = generate_time_series(10000, n_steps + 10)

# Split them 7,000 : 2,000 : 1,000
X_train, y_train = series[:7000, :n_steps],
    series[:7000, -10:, 0] #(7000, 50, 1), (7000,10)
X_valid, y_valid = series[7000:9000, :n_steps],
    series[7000:9000, -10:, 0]
X_test, y_test = series[9000:, :n_steps],
    series[9000:, -10:, 0]
```

4.3 Forecasting Several Time Steps Ahead

```
# Sequential model of two hidden RNN layers
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
        input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])

# MSE = 0.008
```

Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
 1. Implementing a simple RNN
 2. Deep RNNs
 3. Forecasting Several Time Steps Ahead
5. Handling long sequences
 1. LSTM cell
 2. GRU cell
6. Exercises

5. Handling Long Sequences

- Training long sequences has two major challenges:
 - Unstable gradients
 - Forgetting the first inputs in the sequence
- For the **unstable gradients**:
 - **Does not help**: ReLU activation, batch normalization
 - **Helps**: good parameter initialization, faster optimizers, dropout

```
model = Sequential()  
model.add(layers.SimpleRNN(20, dropout=0.2, recurrent_dropout=0.2,  
                           input_shape=[None, 1]))  
model.add(layers.Dense(1))
```

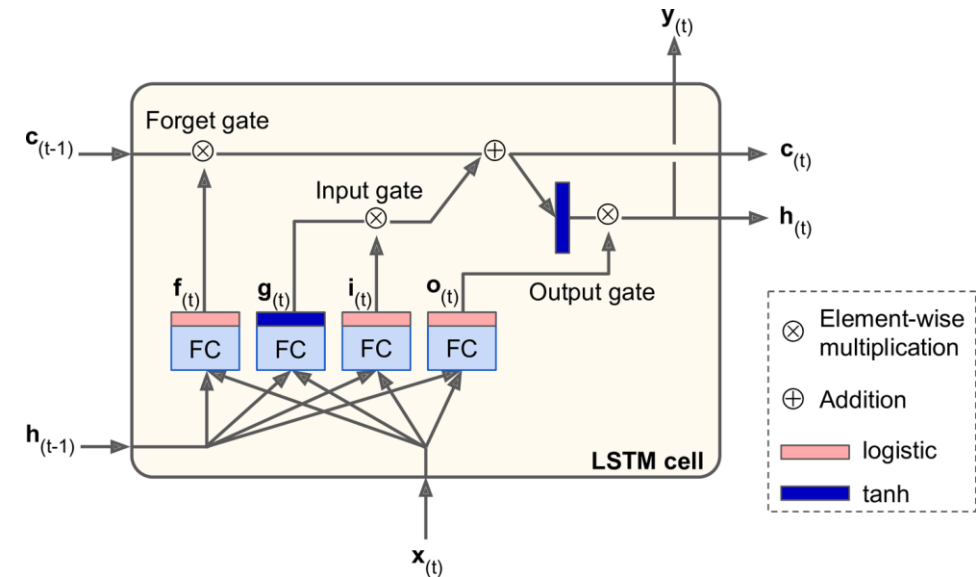
To fight overfitting and
unstable gradients

5. Handling Long Sequences

- To solve the **short-term memory problem**, use
 - **LSTM cell**
 - **GRU cell**
- These cells can be used in place of **SimpleRNN**

5.1 LSTM Cell

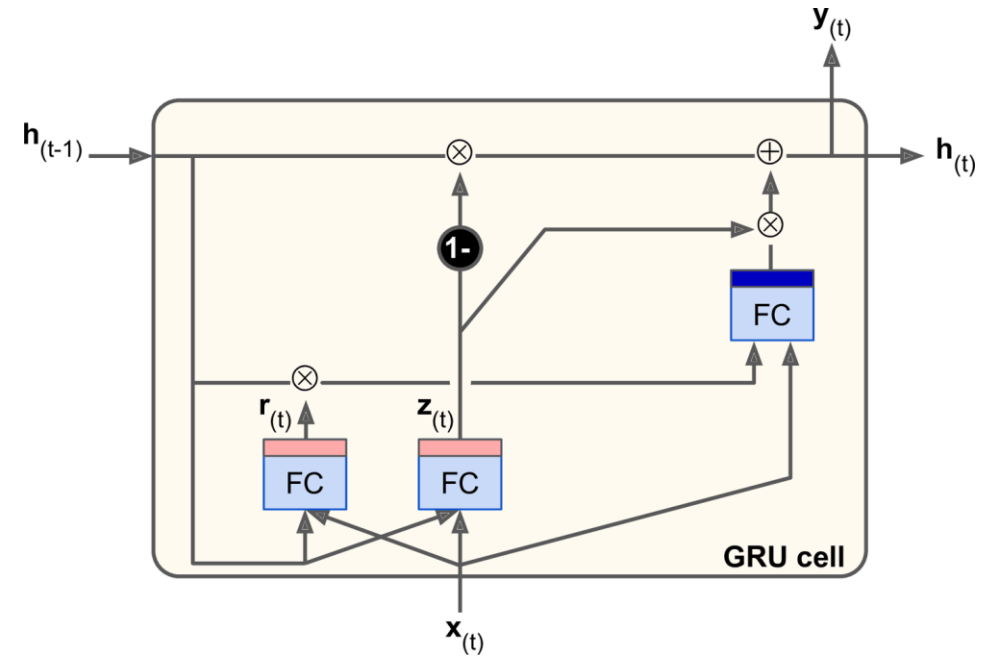
- The **Long Short-Term Memory** (LSTM) cell was proposed in 1997.
- Training converges faster and it **detects long-term dependencies** in the data.
- $h_{(t)}$ as the short-term state and $c_{(t)}$ as the long-term state.



```
model.add(LSTM(20))
```

5.2 GRU Cell

- The **Gated Recurrent Unit** (GRU) cell was proposed in 2014.
- **Simplified version** of the LSTM cell, performs just as well.
- A single gate controls the forget gate and the input gate.



```
model.add(GRU(20))
```

6. Exercises

- 15.1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?
- 15.2. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?
- 15.3. If you want to build a deep sequence-to-sequence RNN, which RNN layers should have `return_sequences=True`? What about a sequence-to-vector RNN?
- 15.4. Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?
- 15.5. What are the main difficulties when training RNNs? How can you handle them?
- 15.6. Can you sketch the LSTM cell's architecture?

Summary

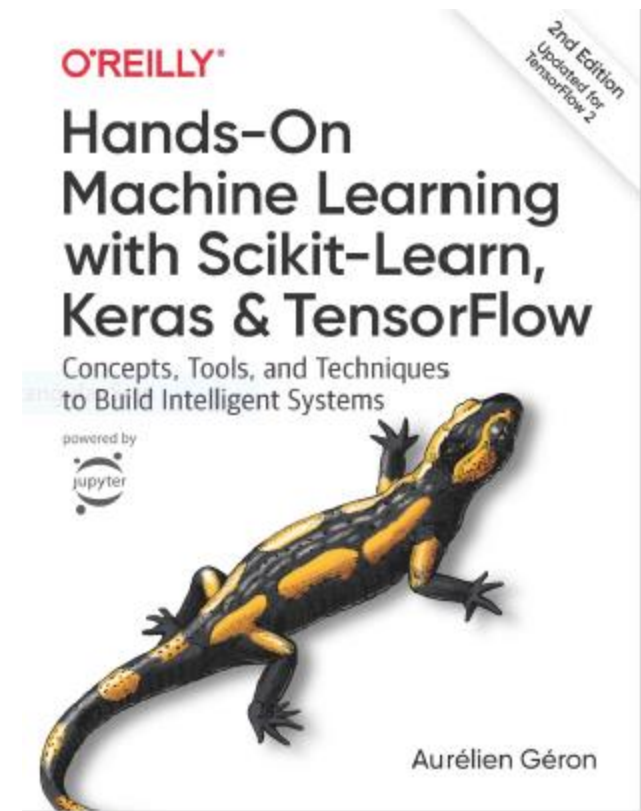
1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
 1. Implementing a simple RNN
 2. Deep RNNs
 3. Forecasting Several Time Steps Ahead
5. Handling long sequences
 1. LSTM cell
 2. GRU cell
6. Exercises

Reinforcement Learning

Prof. Gheith Abandah

Reference

- Chapter 18: **Reinforcement Learning**



- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>

Outline

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policies
5. The Credit Assignment Problem
6. Q-Learning
7. Exercises

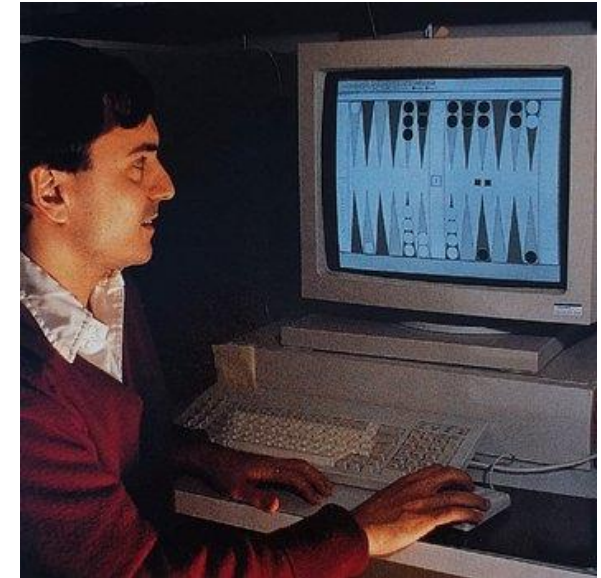
Introduction

- YouTube Video: **An introduction to Reinforcement Learning** from Arxiv Insights

<https://youtu.be/JgvyzIkgxF0>

1. Introduction – History

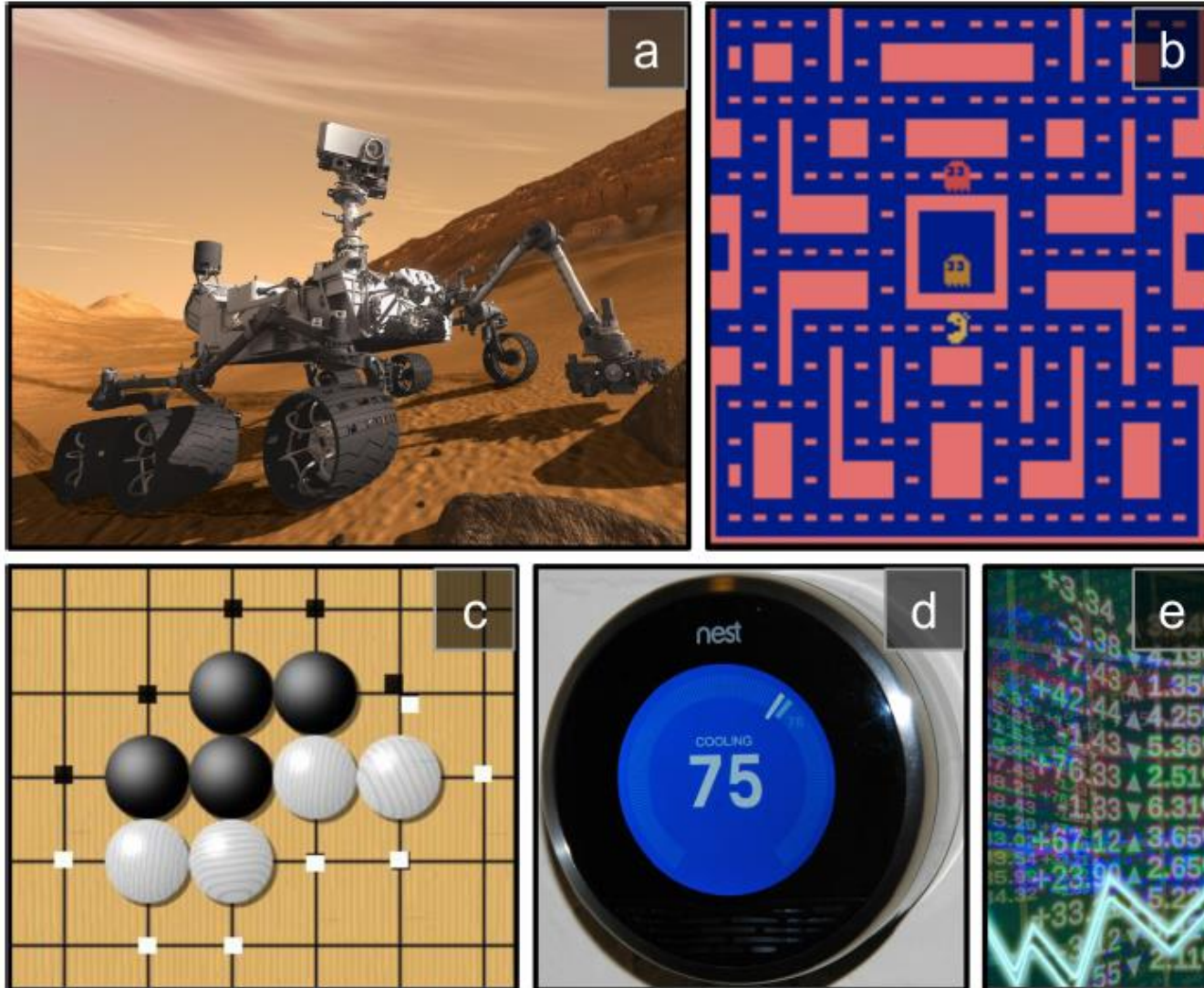
- RL started in **1950s**
- **1992**: IBM's TD-Gammon, a Backgammon playing program.
- **2013**: DeepMind demonstrated a system that learns to play Atari games from scratch.
- Use **deep learning** with raw pixels as inputs and without any prior knowledge of the rules of the games.
- **2014**: Google bought DeepMind for \$500M.
- **2016**: AlphaGo beats Lee Sedol.



1. Introduction – Definition

- In Reinforcement Learning, a software **agent** makes **observations** and takes **actions** within an **environment**, and in return it receives **rewards**.
- Its objective is to learn to act in a way that will **maximize its expected long-term rewards**.
- In short, the agent acts in the environment and learns by trial and error to maximize its **pleasure** and minimize its **pain**.

1. Introduction – Examples



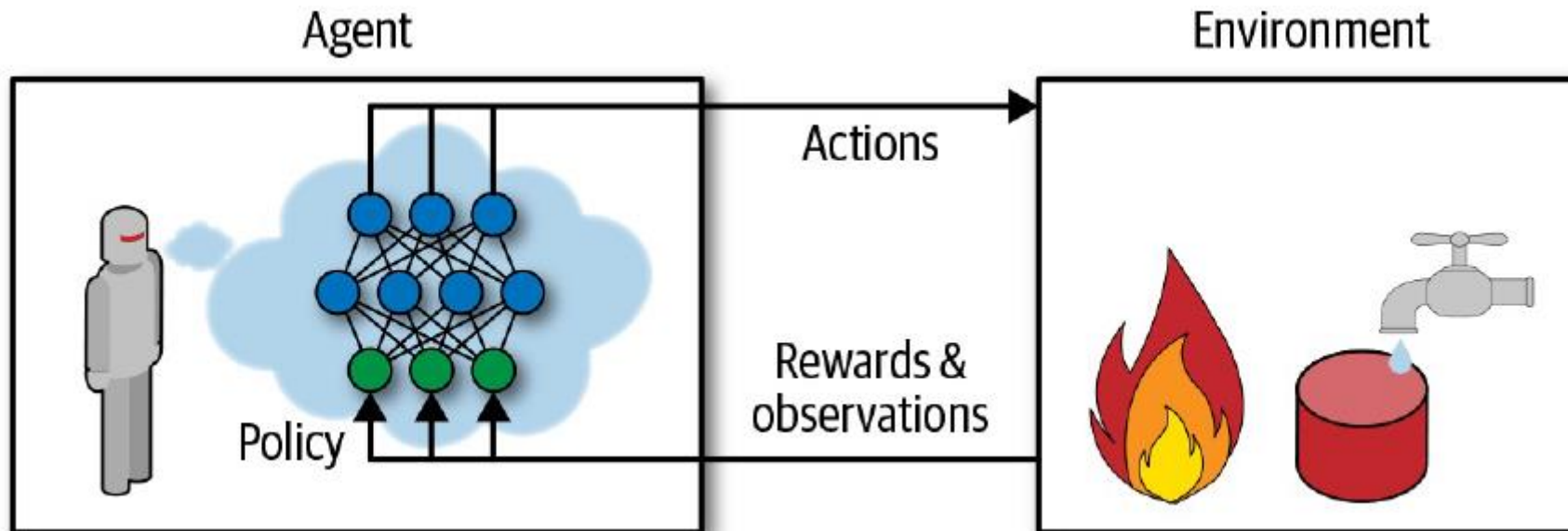
- (a) robotics
- (b) Ms. Pac-Man
- (c) Go player
- (d) thermostat
- (e) automatic trader

Outline

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policies
5. The Credit Assignment Problem
6. Q-Learning
7. Exercises

2. Policy Search

- The algorithm used by the software agent to determine its actions is called its **policy**.
- The policy can be **deterministic** or **stochastic**.
- **Policy search techniques**: Brute force, Genetic algorithm, Policy Gradient (PG), Q-Learning.



Outline

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policies
5. The Credit Assignment Problem
6. Q-Learning
7. Exercises


3. OpenAI Gym

- OpenAI Gym is a toolkit that provides **simulated environments** (Atari games, board games, 2D and 3D physical simulations, ...).
- OpenAI is a nonprofit AI research company funded in part by Elon Musk. Got \$1 billion investment from Microsoft.

```
$ pip3 install --upgrade gym
```

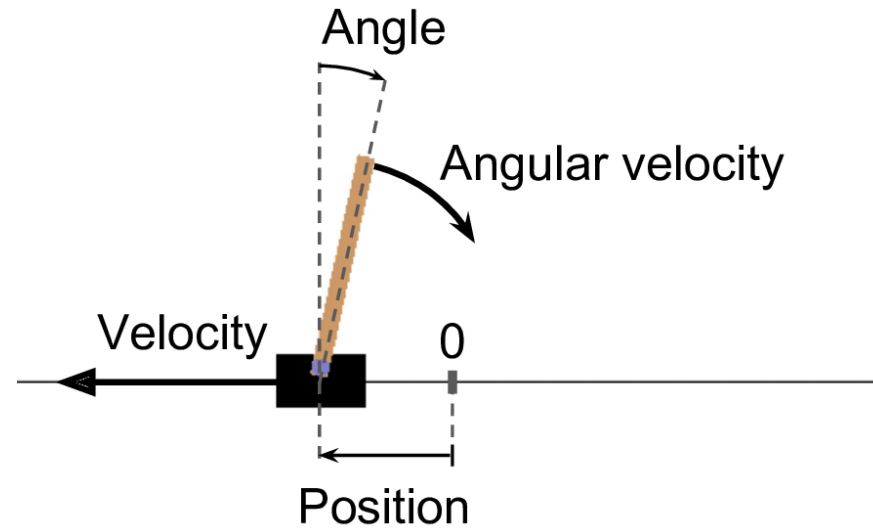
```
>>> import gym
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([-0.012586, -0.001566, 0.042077, -0.001805])
```

Cart position, cart speed,
pole angle, pole velocity



3. OpenAI Gym

```
>>> env.render()
```



- **render()** can also return the rendered image as a NumPy array.

```
>>> img = env.render(mode="rgb_array")
```

```
>>> img.shape # height, width, channels (3 = RGB)
```

```
(800, 1200, 3)
```

3. Balancing the pole

```
>>> env.action_space  
Discrete(2)
```

The possible actions are integers 0 and 1, which represent accelerating left (0) or right (1).

```
>>> action = 1 # accelerate right  
>>> obs, reward, done, info = env.step(action)  
>>> obs  
array([-0.012617, 0.192928, 0.042041, -0.280921])  
>>> reward  
1.0  
>>> done  
False  
>>> info  
{}
```

3. Balancing the pole

```
def basic_policy(obs):  
    angle = obs[2]  
    return 0 if angle < 0 else 1  
  
totals = []  
for episode in range(500):  
    episode_rewards = 0  
    obs = env.reset()  
    for step in range(200):  
        action = basic_policy(obs)  
        obs, reward, done, info = env.step(action)  
        episode_rewards += reward  
        if done:  
            break  
    totals.append(episode_rewards)
```

Accelerates left when the pole is leaning left and accelerates right when the pole is leaning right.

3. Balancing the pole

- Even with 500 tries, this policy never managed to keep the pole upright for more than 68 consecutive steps.

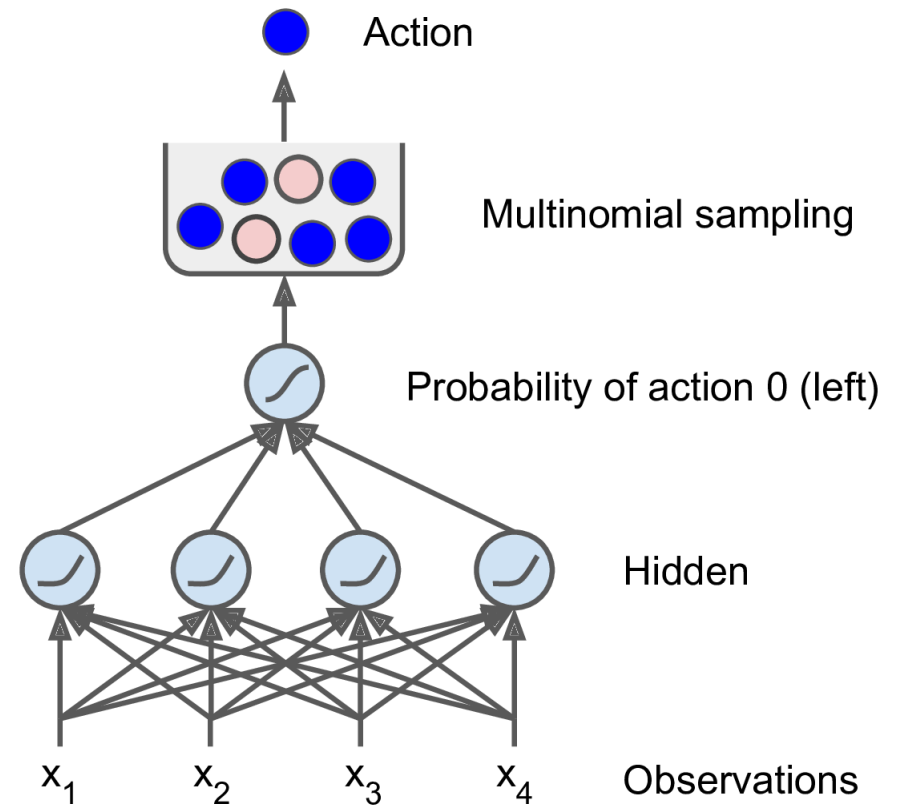
```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals),
      np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

Outline

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policies
5. The Credit Assignment Problem
6. Q-Learning
7. Exercises

4. Neural Network Policies

- Takes an **observation as input**, and **outputs the probability for each action**
- We **select an action** randomly, according to the estimated probabilities.
- **Explore and exploit**



4. Neural Network Policy in Keras

```
# Building a policy network is easy
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

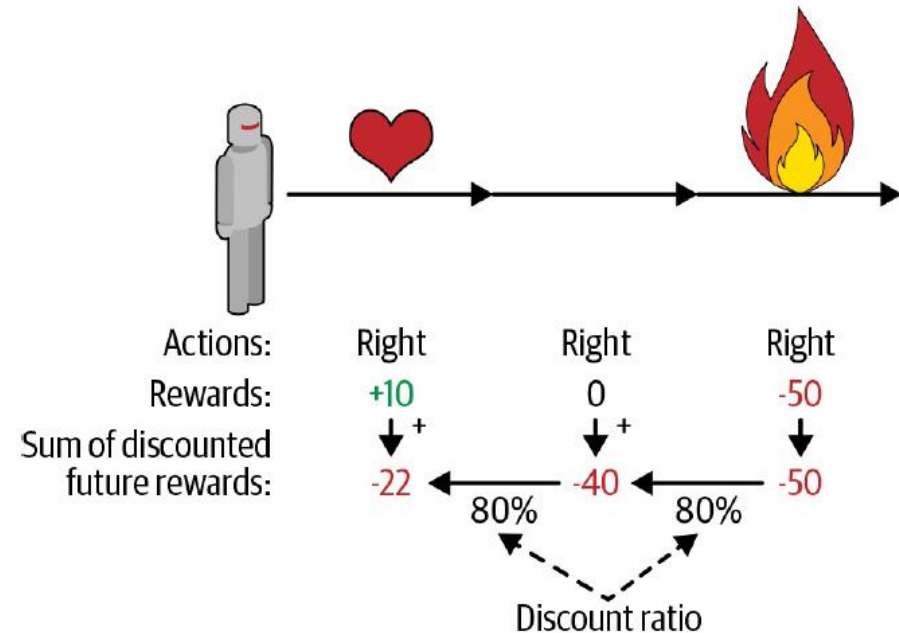
model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu",
                       input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])
# Training it is something else
```


Outline

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policies
5. The Credit Assignment Problem
6. Q-Learning
7. Exercises

5. The Credit Assignment Problem

- Rewards are typically **sparse** and **delayed**.
- **Credit assignment problem:** when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it.
- Evaluate an action based on the sum of all the rewards that come after it, usually applying a **discount rate γ** at each step.



Outline

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policies
5. The Credit Assignment Problem
- 6. Q-Learning**
- 7. Exercises**

6. Q-Learning

- Reference: Keon Kim, Deep Q-Learning with Keras and Gym, <https://keon.io/deep-q-learning/>
- Deep reinforcement learning (deep Q-learning) example to play a CartPole game using Keras and Gym.
- Google's DeepMind published [Playing Atari with Deep Reinforcement Learning](#) where they introduced the algorithm **Deep Q Network** (DQN) in 2013.
- In **DQN**, the **quality function** Q is used to approximate the reward based on a state. $Q(s, a)$ calculates the expected future value from state s and action a .
- A neural network is used to approximate the reward based on the state.

6. Q-Learning

- Carry out an action a , and observe the reward r and resulting new state s' .
- Calculate the maximum target Q and then discount it so that the future reward is worth less than immediate reward by γ .
- Add the current reward to the discounted future reward to get the target value.
- Subtracting our current prediction from the target gives the loss.
- Squaring this value allows us to punish the large loss value more and treat the negative values same as the positive values.

$$\text{loss} = \left(\underbrace{r + \gamma \max_{a'} \hat{Q}(s', a')}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}} \right)^2$$

The equation shows the loss calculation. The term $r + \gamma \max_{a'} \hat{Q}(s', a')$ is labeled as the Target, and $Q(s, a)$ is labeled as the Prediction. Arrows point from the labels 'Reward' and 'Decay Rate' to r and γ respectively.

6. DQN – Imports and Definitions

```
import random
import gym
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
```

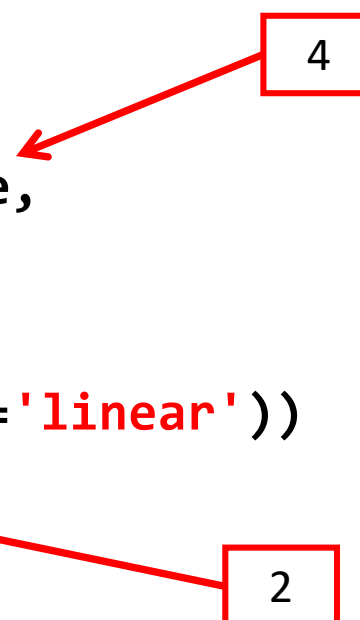
```
EPISODES = 5000
```

6. DQN – Agent Class (1/4)

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01 # min exploration rate
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()
```

6. DQN – Agent Class (2/4)

```
def _build_model(self):  
    model = Sequential()  
    model.add(Dense(24, input_dim=self.state_size,  
                    activation='relu'))  
    model.add(Dense(24, activation='relu'))  
    model.add(Dense(self.action_size, activation='linear'))  
    model.compile(loss='mse',  
                  optimizer=Adam(lr=self.learning_rate))  
    return model
```



6. DQN – Agent Class (3/4)

```
def remember(self, state, action, reward, next_state, done):  
    # Queue of previous experiences to re-train the model  
    self.memory.append((state, action, reward, next_state, done))  
  
def act(self, state):  
    # Returns an action randomly or from the model  
    if np.random.rand() <= self.epsilon:  
        return random.randrange(self.action_size)  
    act_values = self.model.predict(state)  
    return np.argmax(act_values[0])
```

6. DQN – Agent Class (4/4)

```
def replay(self, batch_size):
    minibatch = random.sample(self.memory, batch_size)
    for state, action, reward, next_state, done in
        minibatch:
            target = reward
            if not done:
                target = (reward + self.gamma * np.max(
                    self.model.predict(next_state)[0]))
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1,
                verbose=0)
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
```

Replay() trains the neural net with experiences in the memory

$$loss = \left(r + \gamma \max_a \hat{Q}(s, a) - Q(s, a) \right)^2$$

Learn to predict the reward

6. DQN – Setup

```
if __name__ == "__main__":  
    env = gym.make('CartPole-v1')  
    state_size = env.observation_space.shape[0] # 4  
    action_size = env.action_space.n # 2  
    agent = DQNAgent(state_size, action_size)  
    done = False  
    batch_size = 32
```

6. DQN – Training

```
for e in range(EPIISODES):
    state = env.reset()
    state = np.reshape(state, [1, state_size])
    for time in range(5000):
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        reward = reward if not done else -10
        next_state = np.reshape(next_state, [1, state_size])
        agent.remember(state, action, reward, next_state, done)
        state = next_state
    if done:
        print("episode: {}/{}, score: {}".format(e, EPIISODES, time))
        break
    if len(agent.memory) > batch_size:
        agent.replay(batch_size)
```

6. DQN – Results

```
episode: 1/5000, score: 27  
episode: 2/5000, score: 11  
episode: 3/5000, score: 34  
episode: 4/5000, score: 33  
episode: 5/5000, score: 8  
episode: 6/5000, score: 22  
episode: 7/5000, score: 47  
episode: 8/5000, score: 22  
episode: 9/5000, score: 54  
episode: 10/5000, score: 16
```



```
episode: 284/5000, score: 1331  
episode: 285/5000, score: 124  
episode: 286/5000, score: 259  
episode: 287/5000, score: 138  
episode: 288/5000, score: 170  
episode: 289/5000, score: 13  
episode: 290/5000, score: 365  
episode: 291/5000, score: 1499  
episode: 292/5000, score: 274  
episode: 293/5000, score: 498  
episode: 294/5000, score: 529  
episode: 295/5000, score: 284  
episode: 296/5000, score: 1355  
episode: 297/5000, score: 911  
episode: 298/5000, score: 1414
```

Exercises

- 18.1. How would you define Reinforcement Learning? How is it different from regular supervised or unsupervised learning?
- 18.2. Can you think of three possible applications of RL that were not mentioned in this chapter?
18. For each of them, what is the environment? What is the agent? What are some possible actions? What are the rewards?
- 18.3. What is the discount factor? Can the optimal policy change if you modify the discount factor?
- 18.4. How do you measure the performance of a Reinforcement Learning agent?
- 18.5. What is the credit assignment problem? When does it occur? How can you alleviate it?
- 18.6. What is the point of using a replay buffer?

Summary

1. Introduction
2. Policy Search
3. OpenAI Gym
4. Neural Network Policies
5. The Credit Assignment Problem
6. Q-Learning
7. Exercises

Recommender Systems

Prof. Gheith Abandah

Reference: *Artificial Intelligence with Python*, by Prateek Joshi, Packt Publishing, 2017.

Outline

1. Introduction
2. The MovieLens dataset
3. Similarity scores
4. Building a collaborative recommendation system
5. Open source Python packages
6. Summary

1. Introduction

- YouTube Video: **Recommendation Systems - Learn Python for Data Science #3** by Siraj Raval

<https://youtu.be/9gBC9R-msAk>

1. Introduction

- A **Recommender System** predicts the likelihood that a user would prefer an item and it recommends items to the user.
- **Examples**
 - Facebook — “People You May Know”
 - Netflix — “Other Movies You May Enjoy”
 - LinkedIn — “Jobs You May Be Interested In”
 - Amazon — “Customer who bought this item also bought ...”
 - Google — “Visually Similar Images”
 - YouTube — “Recommended Videos”

1. Introduction

- **Recommender System Types**

1. A **collaborative filtering** algorithm works by finding a set of people with preferences or tastes similar to the target user. Using this smaller set of “similar” people, it constructs a ranked list of suggestions.
2. **Content-based filtering** is based on a description of the item and a profile of the user’s preferences to recommend items that are similar to those that a user liked.
3. **Hybrid**

2. The MovieLens DataSet

- 100,000 ratings (1-5) from 943 users on 1682 movies.
- Includes users data and ratings data

(943, 5) Users

	user_id	age	sex	occupation	zip_code
0	1	24	M	technician	85711
1	2	53	F	other	94043
2	3	23	M	writer	32067
3	4	24	M	technician	43537
4	5	33	F	other	15213

(100000, 4) Ratings

	user_id	movie_id	rating	unix_timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596

3. Similarity Scores

1. **Euclidean score** (Euclidean distance, lower is better)

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

2. **Pearson score** (1 is best)

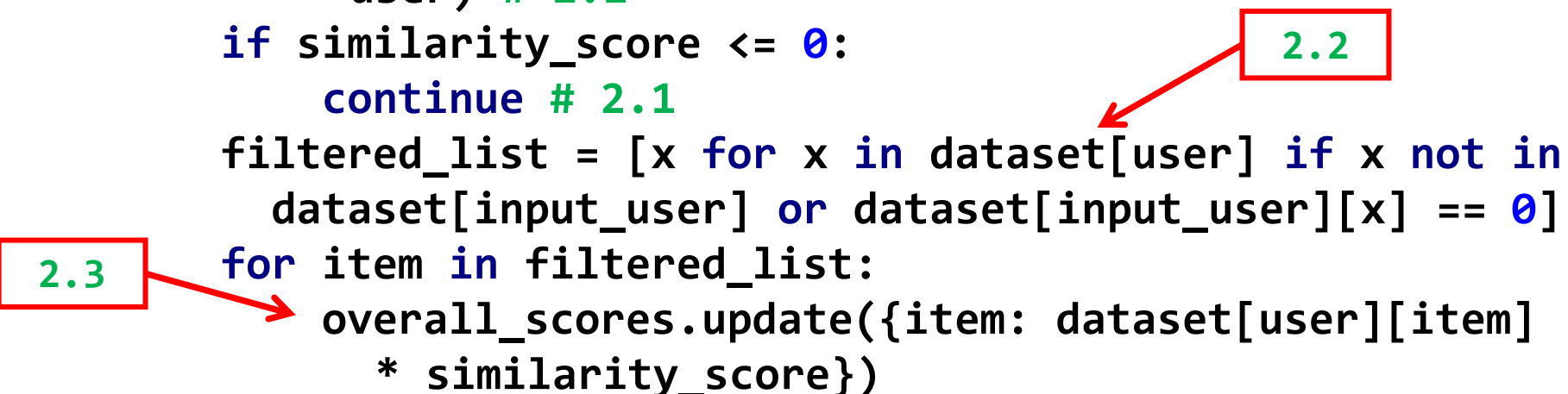
$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

4. Building a Collaborative Recommendation System

1. Function to **recommend movies** for a user
2. **For each other user:**
 1. Find the **Pearson score of commonly rated movies**, ignoring dissimilar users.
 2. Extract a list of movies that have been **rated by this user** but **haven't been rated by the input user**.
 3. For each item in this list, keep a track of the **weighted rating** based on the similarity score.
3. Finally, **sort** the scores and **extract** the **movie recommendations**.

4. Building a Collaborative Recommendation System

```
# Get movie recommendations for the input user
# Assume the input user is in the dataset
# and there is at least one recommendation
def get_recommendations(dataset, input_user): # 1
    overall_scores = {}
    similarity_scores = {}
    for user in [x for x in dataset if x != input_user]:
        similarity_score = pearson_score(dataset, input_user,
            user) # 2.1
        if similarity_score <= 0:
            continue # 2.1
        filtered_list = [x for x in dataset[user] if x not in
            dataset[input_user] or dataset[input_user][x] == 0]
        for item in filtered_list:
            overall_scores.update({item: dataset[user][item]
                * similarity_score})
```



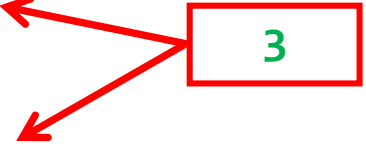
4. Building a Collaborative Recommendation System

```
# Generate movie ranks
movie_scores = np.array([[score, item] for item, score in
                        overall_scores.items()])

# Sort in decreasing order
movie_scores = movie_scores[
    np.argsort(movie_scores[:, 0])[::-1]]

# Extract the movie recommendations
movie_recommendations = [movie for _, movie in
                        movie_scores]

return movie_recommendations
```



5. Open Source Python Packages

- [LightFM](#)
- [GraphLab](#)
- [Crab](#)
- [Surprise](#)
- [Python Recsys](#)
- [MRec](#)

Summary

1. Introduction
2. The MovieLens dataset
3. Similarity scores
4. Building a collaborative recommendation system
5. Open source Python packages
6. Summary