

Data-intensive space engineering

Lecture 1

Carlos Sanmiguel Vila

Instructor

Lecturer: INTA – Dr. Carlos Sanmiguel Vila

Short Bio:

- **Education:** Aerospace Eng. MSc'01 & BSc'14 (UPM, SP),
(10yrs) Fluid Mechanics. PhD'19 (UC3M, USA)

- **Academic Experience:** 1yrs Postdoc (UC3M, SP),
(3yrs) 2yrs Adjunct Professor (UC3M, SP)

- **Industrial Experience:** 2yrs Data Scientist/Analyst (Santander, UK),
(5yrs) 3yrs Research Scientist (INTA, SP)

- **Expertise & projects:** Artificial intelligence in the aerospace industry, applications in Surrogate Model for Aerodynamics (aircraft), flow control, reduced-order modelling, multifidelity and super resolution algorithms.

Author of over 20 publications (peer-reviewed journals and conferences). **Research Team and/or Principal Investigator of 20 projects**, in aeronautics (TIFON as PI).



Mail: csanmigu@ing.uc3m.es

C.Sanmiguel Vila

Course Information

- **Course Aim:** Explore statistical and artificial intelligence techniques for the analysis of space-engineering data via:
- Theoretical sessions: Lectures will introduce key concepts, theories, and the latest developments in data-intensive techniques, laying a solid foundation for students
- Practical sessions: hands-on laboratory sessions where students will apply the discussed machine learning techniques to real-world space engineering problems using computer-based tools

Course Information

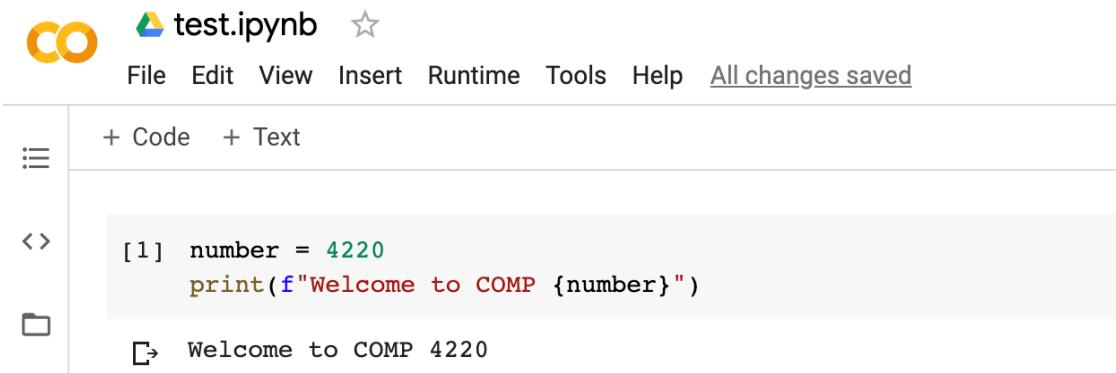
Practical sessions:

- Python will be the programming language during the course.
- Labs will be Google Colab notebooks, which will count as 75% of the final grade.
- End-of-term-examination will be 25% of the final grade.

Google Colab

Google Colaboratory, or “Colab” for short, allows you to write and execute Python in your browser

- Zero configuration required
- Free access to GPUs and other computing resources
- Use terminal commands on Google Colab
- <https://drive.google.com> (for the first time, select “Connect more apps”)



The screenshot shows the Google Colab interface. At the top, there's a toolbar with icons for file operations, a file named "test.ipynb", and a star icon. Below the toolbar is a menu bar with File, Edit, View, Insert, Runtime, Tools, Help, and a status message "All changes saved". The main area has two tabs: "+ Code" and "+ Text". A code cell contains the following Python code:
[1] number = 4220
print(f"Welcome to COMP {number}")
The output of the cell is:
Welcome to COMP 4220

<https://www.youtube.com/watch?v=RLYoEylHL6A>

Course Information

To pass the course, the following two requirements must be met:

- Obtain a MINIMUM of 4.0/10 in the final exam
- Obtain a MINIMUM of 5.0/10 in the overall grade (obtained weighting 25% of the final exam and 75% of the continuous evaluation)

The continuous evaluation includes 5 laboratory sessions with corresponding reports (each corresponding to 15% of the final grade)

Continuous evaluation

Continuous evaluation will consist of:

- Complete a Google Colab Python notebook proposed during laboratory class.
- Prepare a presentation of 5 minutes about a scientific paper related to the class topics.
- 2 students will form groups.

Continuous evaluation

The presentations will consist of a few slides, where the students will look for a practical application of machine learning in the aerospace industry.

Students must ask the following questions:

- What is the problem that is investigated?
- Which is the dataset employed? Describe it briefly.
- What are the main results obtained?

Continuous evaluation

To look for scientific articles, students can use:

- <https://scholar.google.es/> (Use UC3M VPN)
- <https://www.perplexity.ai/>
- <https://elicit.com/>

Take care using ChatGPT since it uses fake references and made-up studies.

Students must send me the selected paper in advance to avoid coincidences between groups. The articles will be selected on a first-come, first-served basis; if you have troubles, you can ask for help.

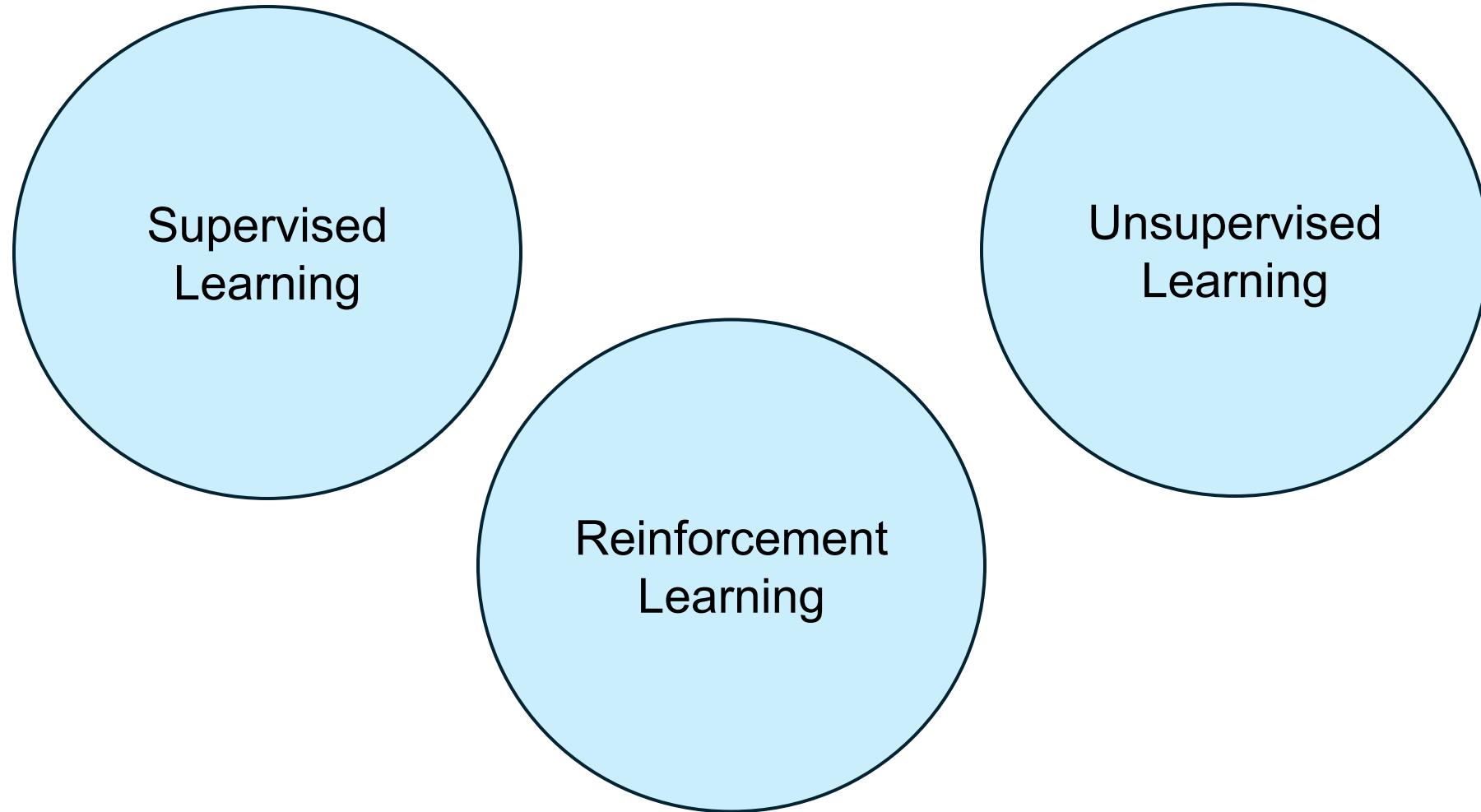
Definition of Machine Learning

- Arthur Samuel (1959): Machine Learning is the field of study that gives the computer the ability to learn without being explicitly programmed.
- Tom Mitchell (1998): a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

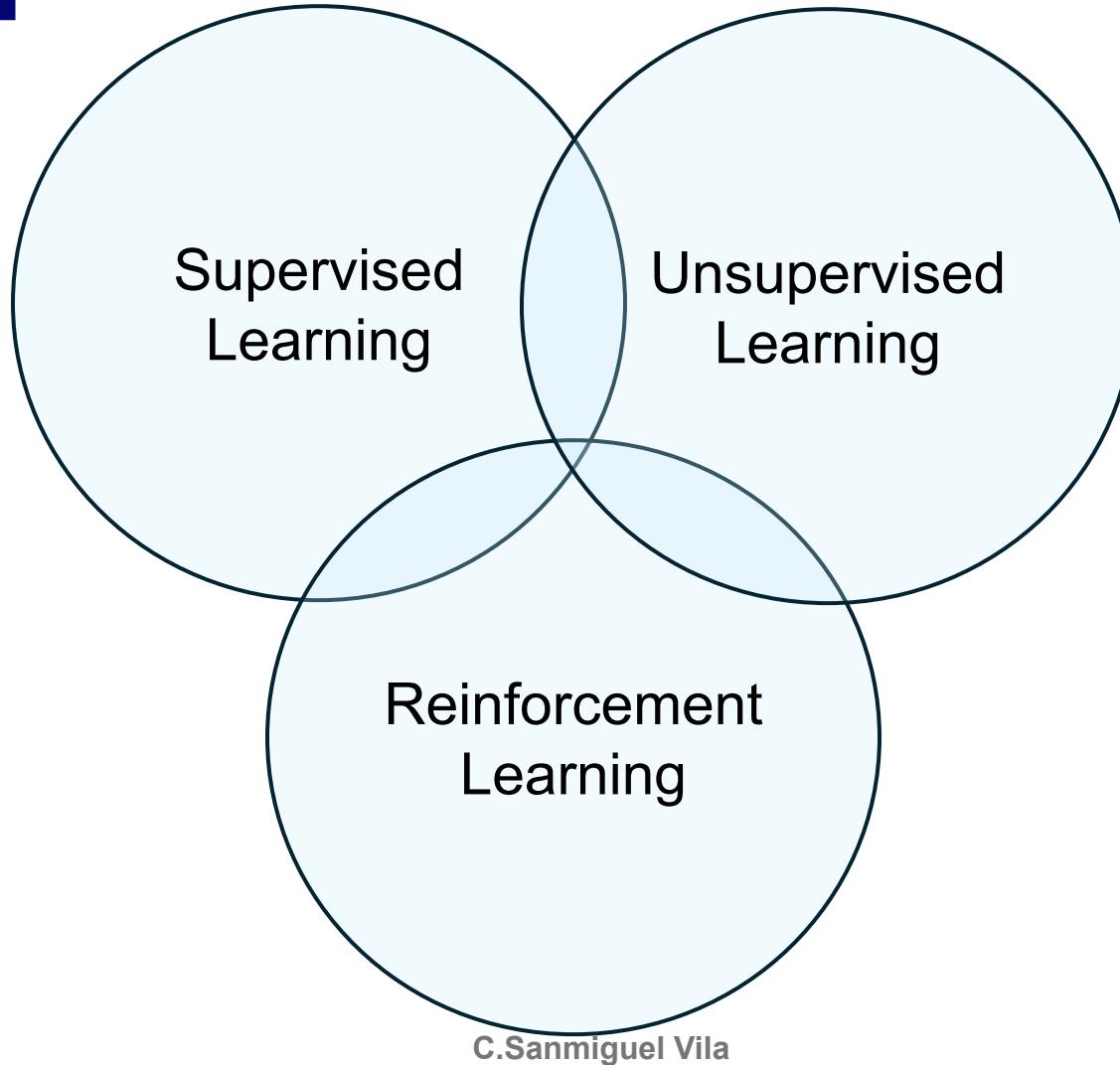
Experience (data): games played by the program (with itself) Performance measure: winning rate



Taxonomy of Machine Learning (A Simplistic View Based on Tasks)



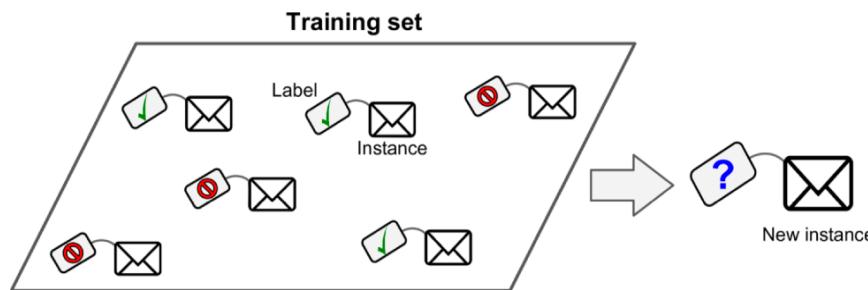
Taxonomy of Machine Learning (A Simplistic View Based on Tasks)



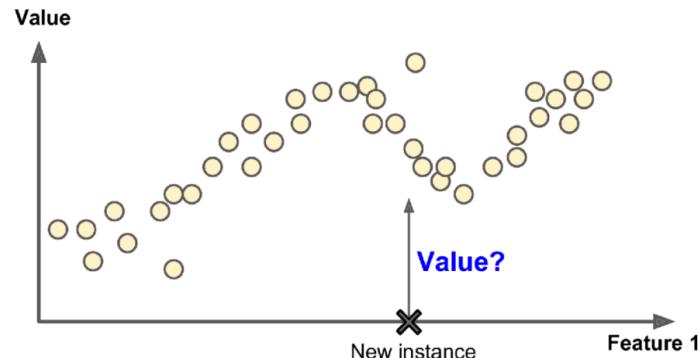
Supervised Learning

Supervised learning: training set includes the desired solutions or labels

- **Classification:** labels are classes (e.g., spam or ham)



- **Regression:** labels are numeric values (e.g., price of car)



Supervised Learning - Regression

1. Exoplanet Detection and Characterization:

Regression models are used to analyze light curves from stars to detect and characterize exoplanets.

2. Solar Flare Prediction:

Regression models are employed to predict the intensity and duration of solar flares based on solar activity data.

3. Space Weather Forecasting:

Regression techniques are used to predict various space weather phenomena, such as geomagnetic storms.

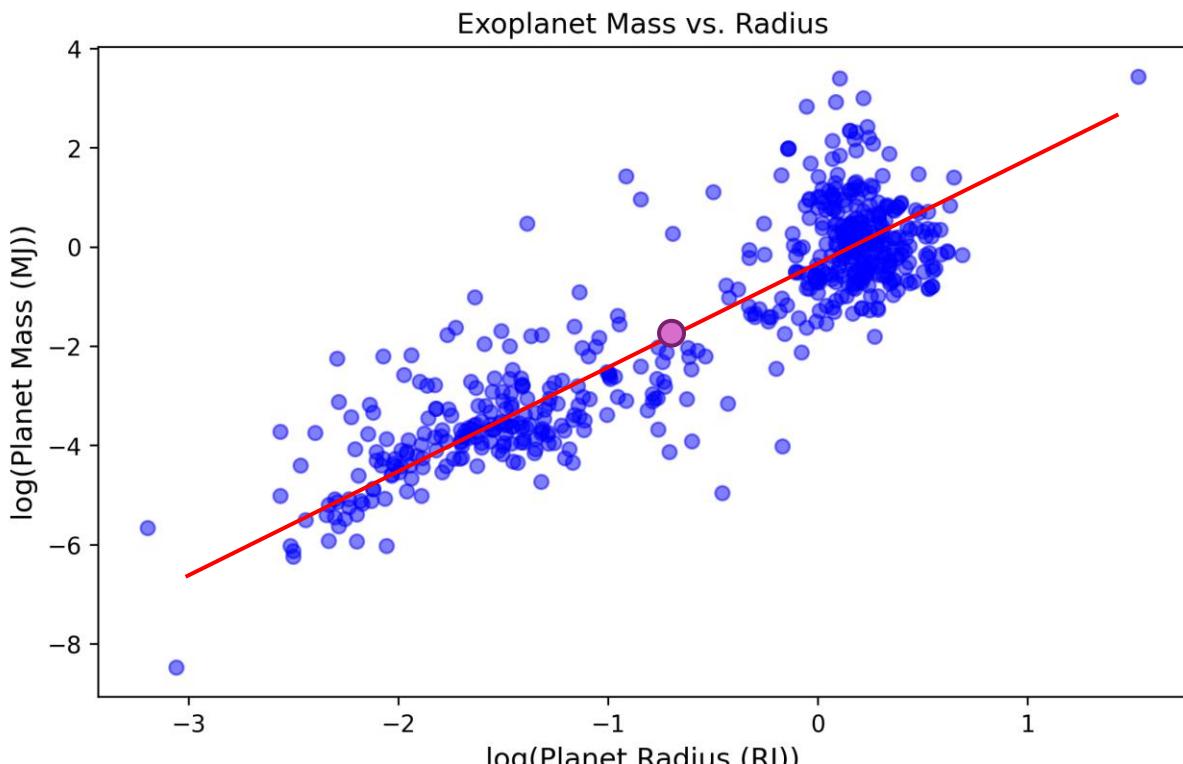
4. Satellite Orbit Prediction:

Regression models are used to predict satellite orbits and potential collisions with space debris.

5. Spectral Analysis of Astronomical Objects:

Regression models are applied to analyze spectral data from stars, galaxies, and other celestial objects.

Supervised Learning - Regression



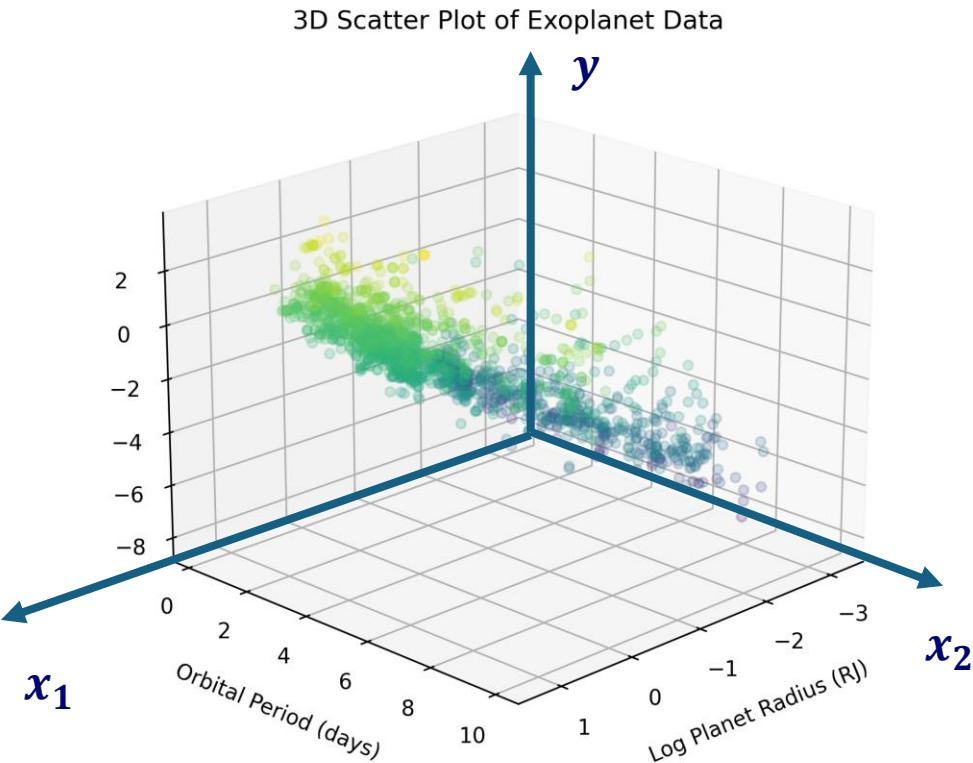
Data from NASA Exoplanet Archive

- Given: a dataset that contains n samples

$$(x^{(1)}, y^{(1)}), \dots (x^{(n)}, y^{(n)})$$

Task: if a planet has x radius, predict its mass?

Supervised Learning - Regression



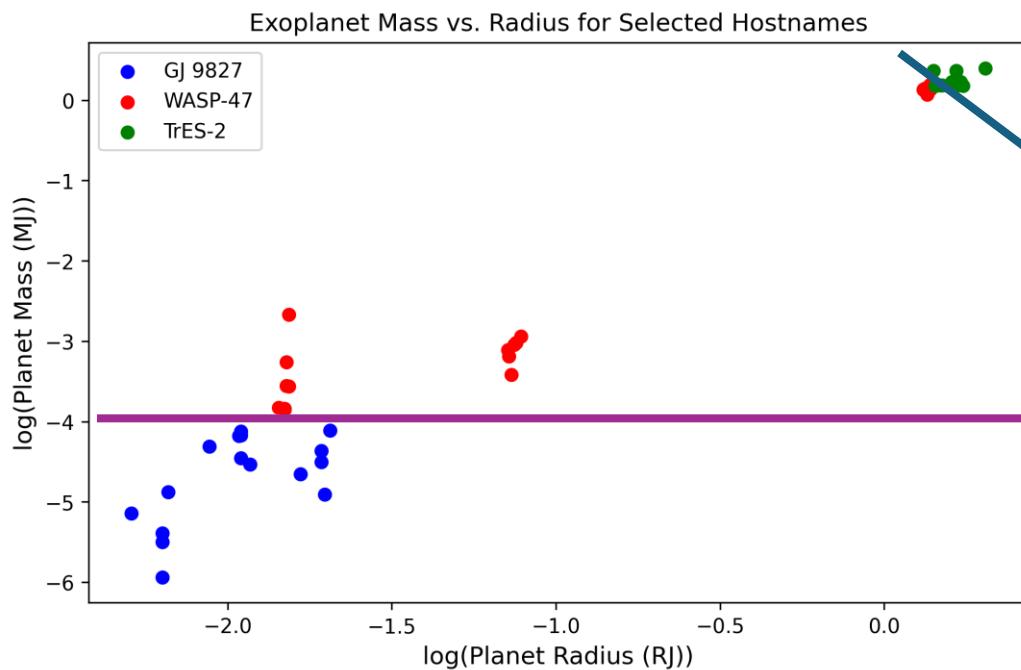
Data from NASA Exoplanet Archive

- Suppose we also know other information
- **Task:** find a function that maps
(orbital period, Planet radius) \rightarrow (Planet Mass)
↓
features/input
 $x \in \mathbb{R}^2$
- label/output
 $y \in \mathbb{R}$
- Dataset: $(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})$ where in general we can have a very large number of features $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_d^{(i)})$ with $x \in \mathbb{R}^d$
- Supervision refers to $y^1, \dots, y^{(n)}$

Supervised Learning - Classification

- 1. Space Object Identification and Classification:** Extract information from the hyperspectral signatures of unknown space objects. It involves using machine learning techniques to decompose spectra and identify materials, followed by probabilistic classification of space objects based on material identification. Neural Networks(NNs) can be used to decompose and classify satellites into categories such as communication satellites, rocket bodies, and Earth observation satellites.
- 2. Exoplanet Detection:** Classification models are used to identify potential exoplanets from light curve data collected by space telescopes like Kepler. These models classify whether a dip in light intensity is due to an exoplanet transit or other phenomena.
- 3. Astronomical Object Classification:** Classifying celestial objects such as stars, galaxies, and quasars based on their spectral data. This helps in organizing large astronomical datasets and identifying new types of objects.

Regression vs Classification

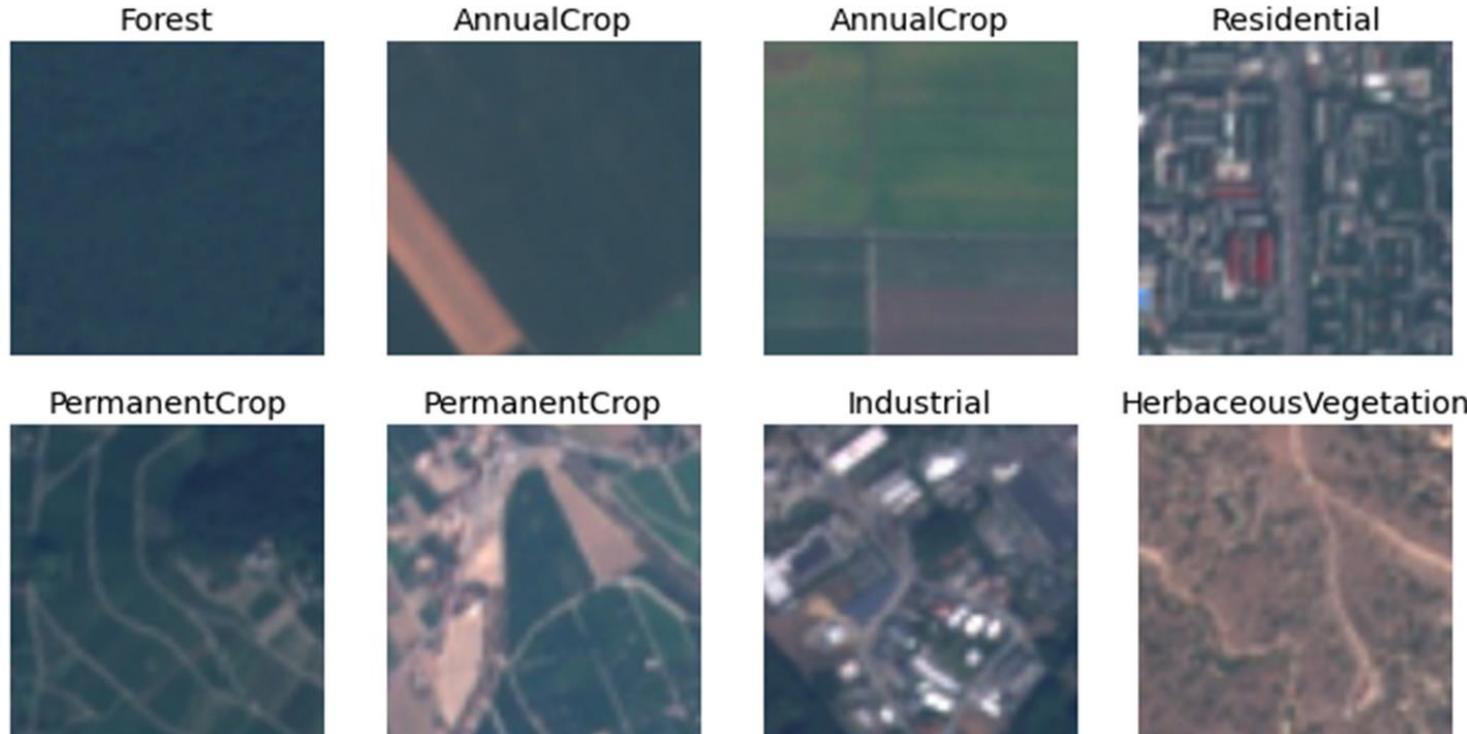


Data from NASA Exoplanet Archive

- **Regression:** if $y \in \mathbb{R}$ is a continuous variable, e.g., Planet mass
- **Classification:** the label is a discrete variable e.g., the task of predicting which is their host star
 $(\text{Planet Mass}, \text{Planet radius}) \rightarrow (\text{Host star})$

Supervised Learning in Computer Vision

- Image Classification
- x = raw pixels of the image, y = the main object



Sample images from
EuroSAT dataset

Supervised Learning in Computer Vision

- Object localization and detection
- x = raw pixels of the image, y = the bounding boxes

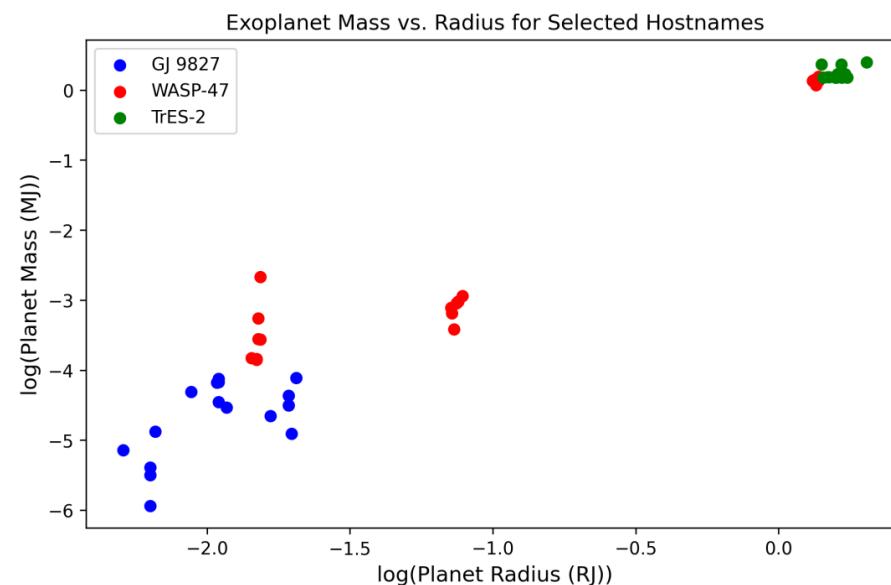


Sample images
from the [PlanesNet](#)
dataset

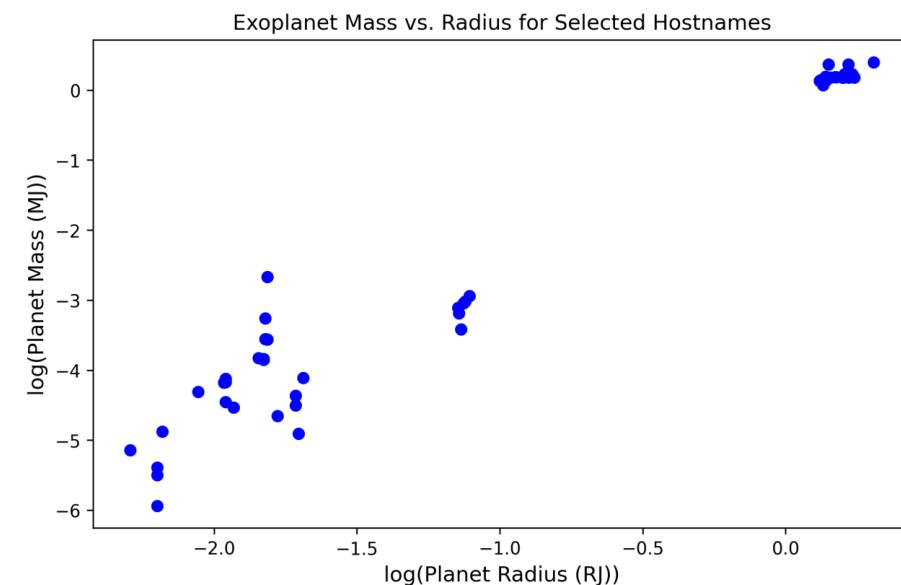
Unsupervised Learning

- Dataset contains no labels: $(x^{(1)}, \dots, x^{(n)})$
- Goal (vaguely-posed): to find interesting structures in the data

Supervised



Unsupervised

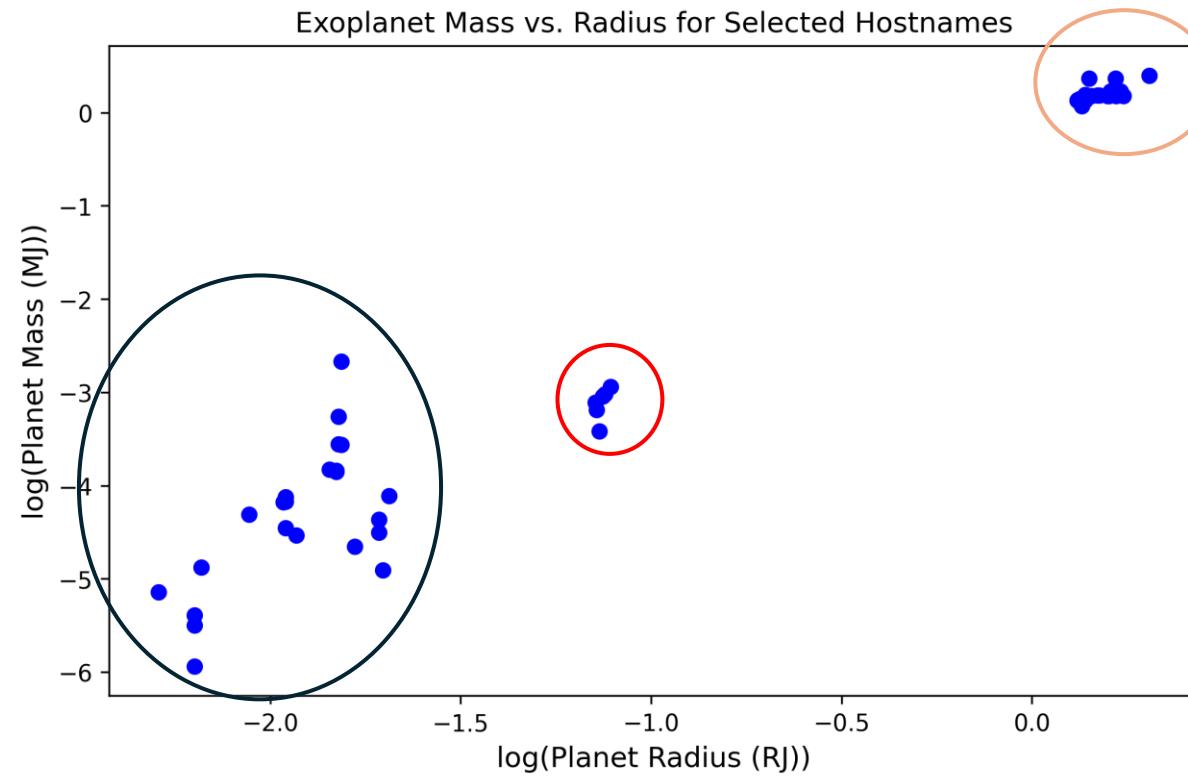


Unsupervised Learning

- **Astronomical Object Classification:** Unsupervised learning techniques are used to classify celestial objects like stars, galaxies, and quasars based on their spectral data
- **Galaxy Morphology Classification:** Unsupervised methods are applied to classify galaxies based on their morphology without relying on predefined categories
- **Exoplanet Detection:** Unsupervised learning is used to analyze light curve data from space telescopes to detect potential exoplanets
- **Space Object Identification:** Clustering techniques are used to identify and classify space objects based on their spectral signatures
- **Solar Wind Classification:** Unsupervised learning is applied to classify different regions of the Earth's magnetosphere based on solar wind parameters

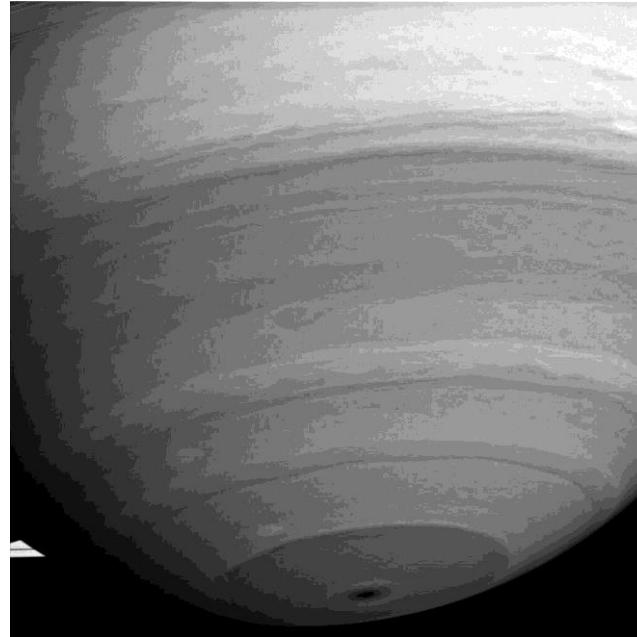
Unsupervised Learning - Clustering

Clustering: Detect groups of similar data



Unsupervised Learning – Dimensionality Reduction

Dimensionality Reduction: Reduce the dimension of the data, i.e., image compression



<https://science.nasa.gov/resource/image-compression/>

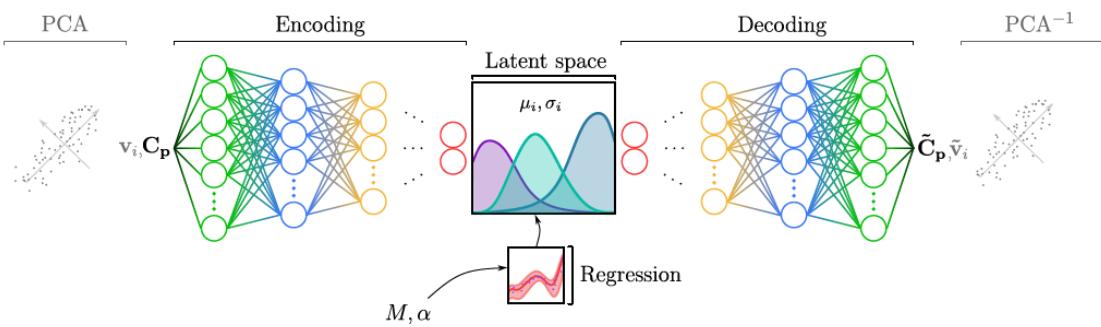
Unsupervised Learning – Dimensionality Reduction

Anomaly detection: find new instances that look different



Supervised & Unsupervised Learning

Reduce dimensionality to perform a more efficient regression in a low-dimensional space



FRANCÉS-BELDA, Víctor, et al. Towards aerodynamic surrogate modeling based on β -variational autoencoders. arXiv preprint arXiv:2408.04969, 2024.

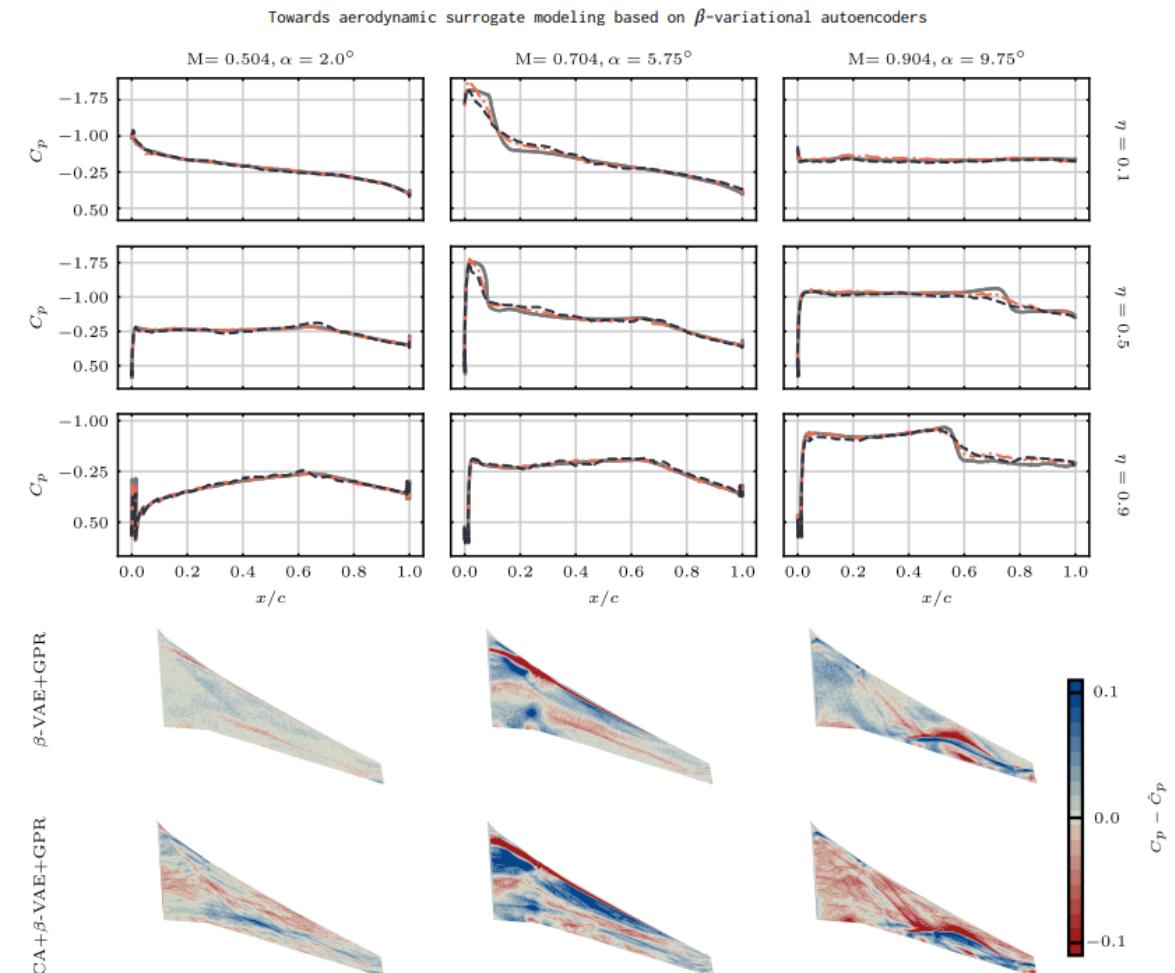
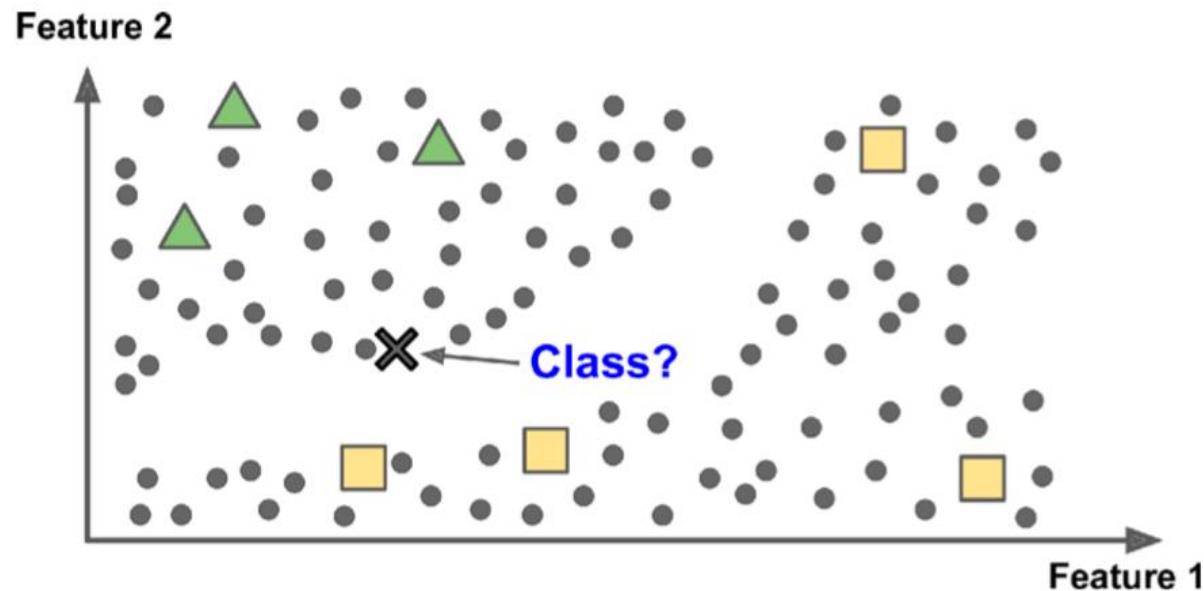


Figure 12: Difference between ground truth and predicted coefficients for the visualization test cases with $\beta = 0.008$. Chordwise pressure distributions from fine-tuned β -VAE+GPR (---) and $\text{PCA}+\beta$ -VAE+GPR (-·-) models at span percentages $\eta = 0.1, 0.5, 0.9$ are displayed and contrasted with ground truth (—).

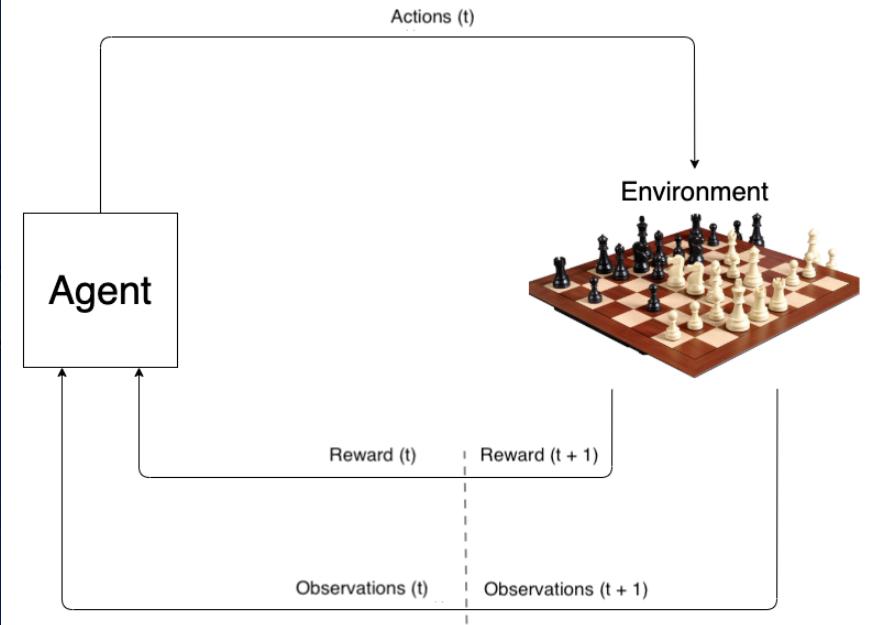
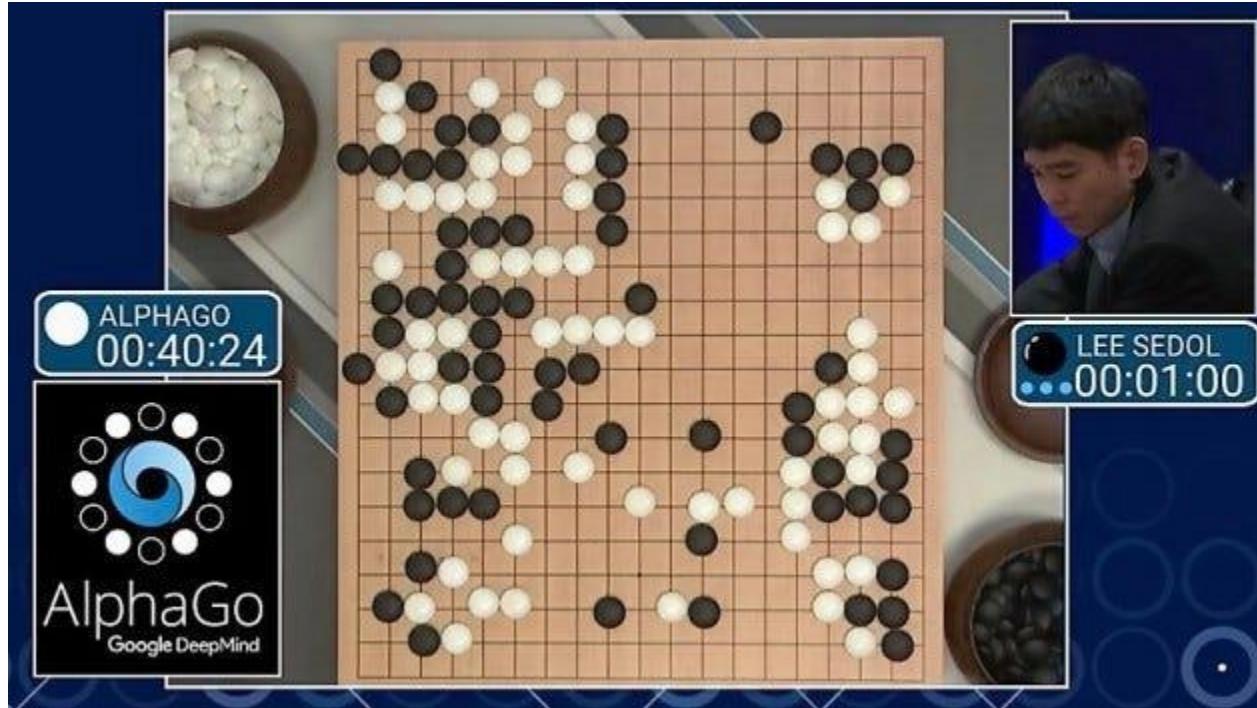
Semisupervised Learning

Semisupervised learning: plenty of unlabeled instances and few labeled



Reinforcement Learning

Learning to make sequential decisions.



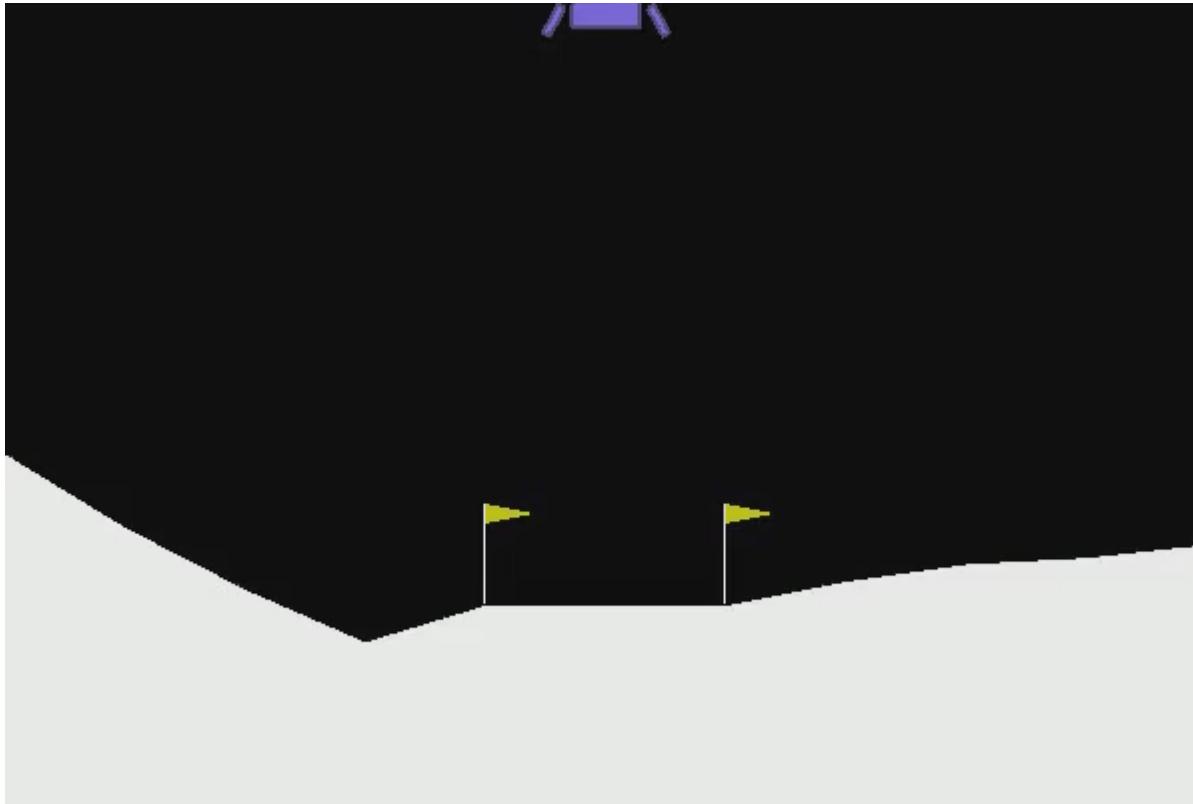
<https://youtu.be/3jDoPobFgwA>

C.Sanmiguel Vila

28

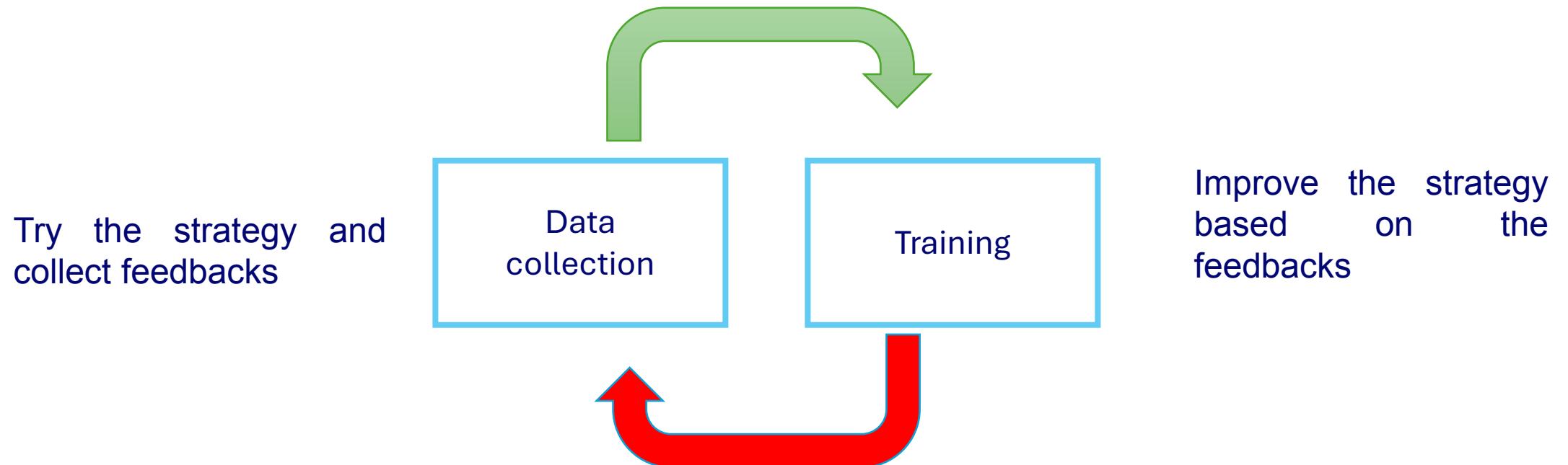
Reinforcement Learning

Learning to make sequential decisions.



Reinforcement Learning

The algorithm can collect data interactively



Natural Language Processing - Large Language Models

Can NLP improve the efficiency of space mission planning?

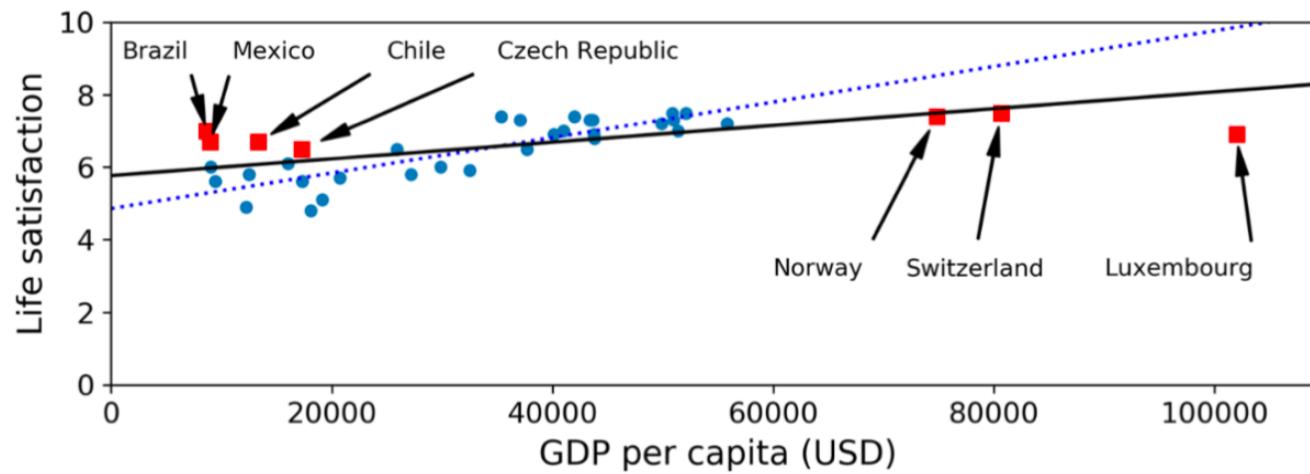
1. Yes, Natural Language Processing (NLP) can significantly improve the efficiency of space mission planning in several ways:
Requirements Management: NLP can extract semantic information from textual requirements in space engineering documents.
 2. It can help find related requirements, enhancing search capabilities in requirement databases.
 3. NLP can check consistency of traceability between requirements and identify suspicious or overlapping requirements.
 4. Knowledge Reuse: ESA has developed a method using NLP to automatically extract critical information from space mission data to populate a Knowledge Graph.
 5. This unified view of the space ecosystem enables mission experts to find unstructured information more efficiently.
 6. It allows users to query all knowledge from a single interface instead of reading through thousands of web pages.
 7. Information Extraction: NLP can parse data from unstructured datasets related to space missions.
 8. It can extract key information about mission requirements, objectives, payloads, propulsion systems, and target environments from various documents.
 9. Communication Enhancement: NLP enables spacecraft to comprehend and respond to complex voice commands, aiding astronauts in controlling and monitoring various systems.
 10. It helps translate scientific data into human-readable formats, facilitating seamless collaboration between scientists and engineers.
 11. Data Analysis: NLP techniques can analyze large volumes of space-related text data, such as scientific papers, mission reports, and news articles.
 12. This analysis can uncover trends, relationships, and insights that may not be immediately apparent to human planners.
 13. Predictive Analytics: By analyzing historical mission data and documentation, NLP can help predict potential issues or challenges in future missions.
 14. Automated Documentation: NLP can assist in generating and organizing mission documentation, saving time and ensuring consistency.
- These applications of NLP can significantly streamline the mission planning process, reduce human error, and enable more efficient use of existing knowledge and resources in space mission design and execution.

Machine Learning Pipeline

- Study the data (data preparation)
- Select a model or learning algorithm
- Train on the available data and assess performance
- Apply the model to make predictions on new cases

Challenges of machine learning

- Insufficient quantity of training data
- Non-representative and poor-quality training data

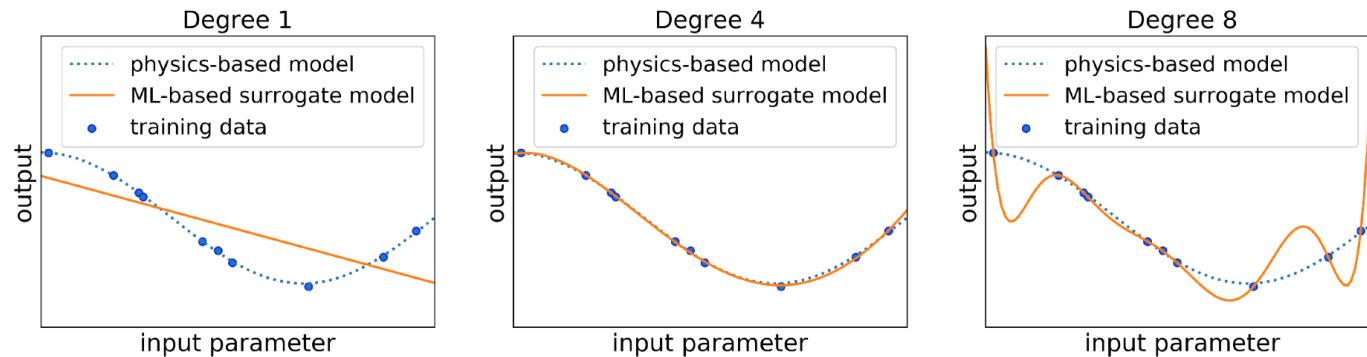


- Irrelevant features: coming up with a good set of features

Challenges of machine learning

Overfitting the training data

- When the model is too complex relative to the amount and noisiness of data



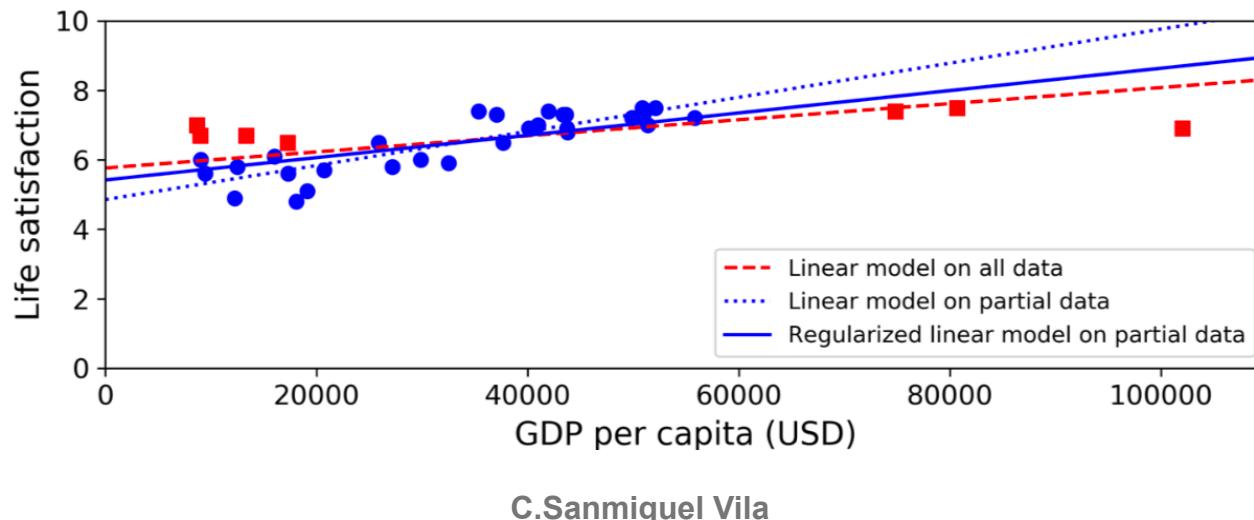
How to alleviate this problem?

- Simplify the model
- Gather more training data
- Reduce noise and outliers (i.e., data preparation)

Challenges of machine learning

Use a subset of data (train, test and validation splits) and regularization

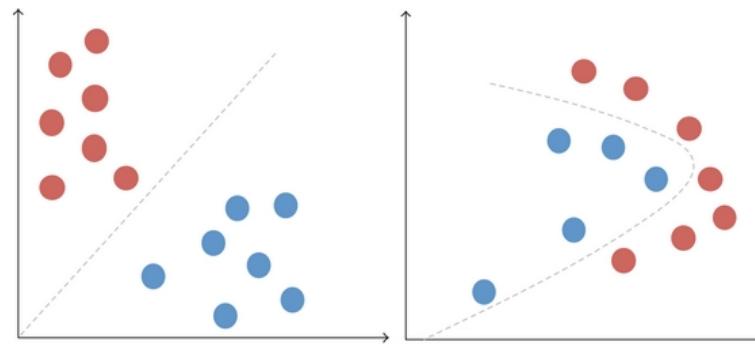
- Regularization: balance between fitting the data perfectly and keeping the model simple enough
- A hyperparameter controls the amount of regularization (must be set prior to learning and remains constant during training)
- Tuning hyperparameters is extremely important



Challenges of machine learning

Underfitting the training data

- When the model is too simple

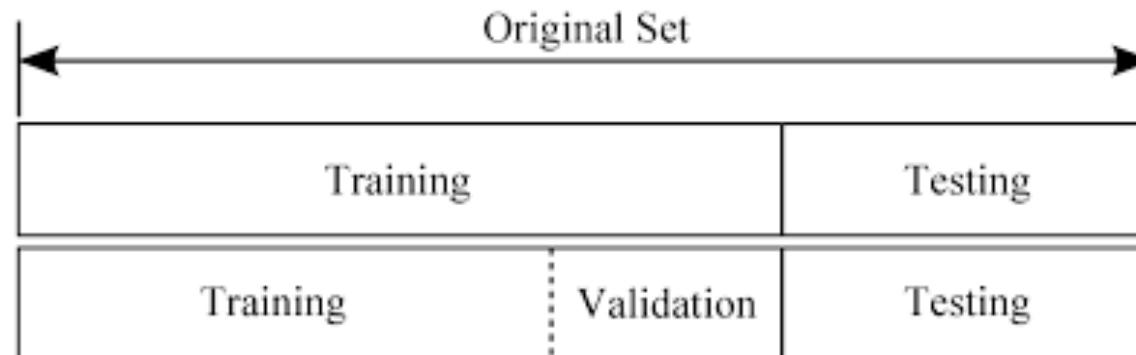


How do we solve this problem?

- Select a more powerful model
- Reduce constraints on the model, such as regularization

Challenges of machine learning

- The only way to know how a model will generalize to new cases is to try it out on new cases
- Split the available data into two main sets: training set and testing set
- The error on the test set is called the generalization error or out-of-sample error
- If the training error is low and the generalization error is high, then overfitting occurs
- Choose suitable models with an adequate set of hyperparameters



Data-intensive space engineering

Lecture 2

Carlos Sanmiguel Vila

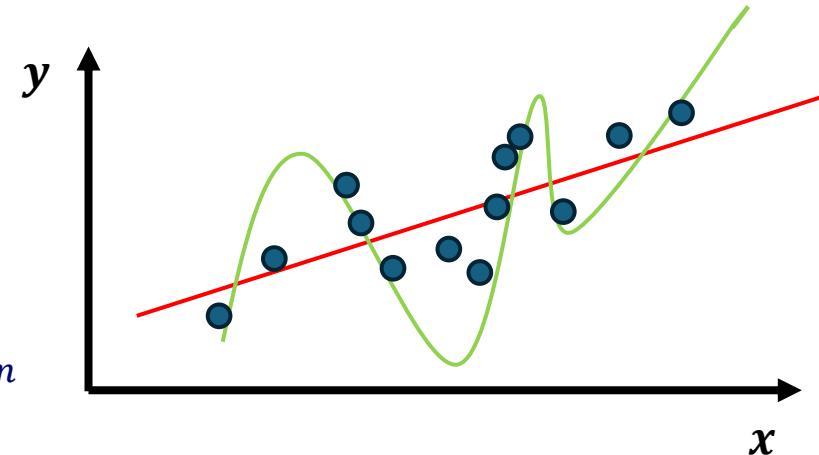
Supervised Learning

Prediction

Given: a training dataset that contains n samples

$$(x^{(1)}, y^{(1)}), \dots (x^{(n)}, y^{(n)})$$

$$x^{(i)} \in X, y^{(i)} \in Y, \text{ being } X \in \mathbb{R}^{n_x \times n}, Y \in \mathbb{R}^{n_y \times n}$$



We want to find a good $h: X \rightarrow Y$ (hypothesis)

We care about new values that are not in our training set

Given a set of unseen points x , which is their y ?

If y is discrete \rightarrow Classification

If y is continuous \rightarrow Regression

light intensity \rightarrow Is this an exoplanet?
Planet radius \rightarrow Planet Mass

Regression

In regression, we want to quantify the relationship between continuous variables and make predictions.

$$h: X \rightarrow Y$$

How do we represent h ?

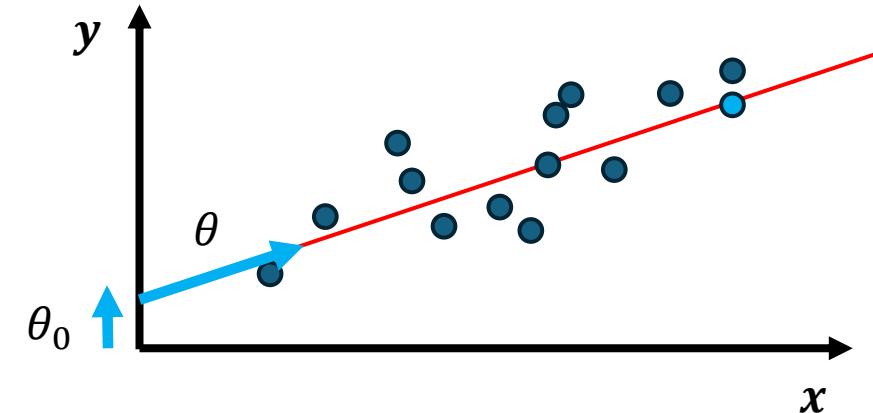
	$x_1^{(1)}$	$x_2^{(1)}$
$x^{(1)}$	Planet Radius	Orbital Period
	1.25	3.101278
$x^{(2)}$	1.24	3.246740

features

$$h(x) = \theta_0 + \theta_1 x$$

	y
....	Planet Mass
	0.62
	1.02

target



$$h(x) = \theta_0 x_0 + \theta_1 x_1 + \dots + \theta_d x_d = \sum_{j=0}^d \theta_j x_j \text{ with } x_0 = 1$$



$$\hat{y} = h(x) = \theta x$$

Regression

$$\hat{y} = h(x) = \theta x$$

$$\begin{aligned}y &= h(x) + \epsilon \\&= \theta x + \epsilon\end{aligned}$$

- h is the hypothesis function, using the model parameters θ
- θ is the model's parameter vector, containing the bias term θ_0 and the feature weights θ_1 to θ_n
- x is the instance's feature vector, with x_0 always equal to 1.
- \hat{y} is the predicted value.
- ϵ is an error term that captures either unmodeled effects (such as if there are some features very pertinent to predicting the target, but that we'd left out of the regression), or random noise

We have a regression... but how do we train it?

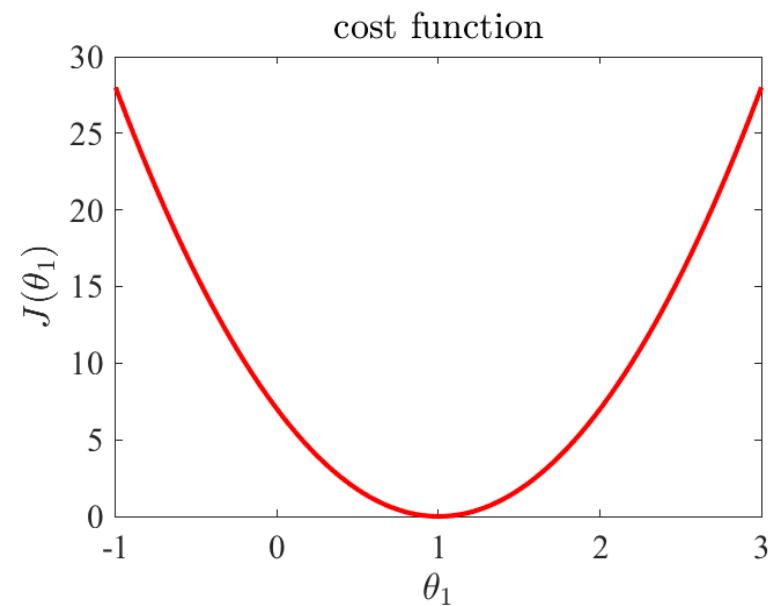
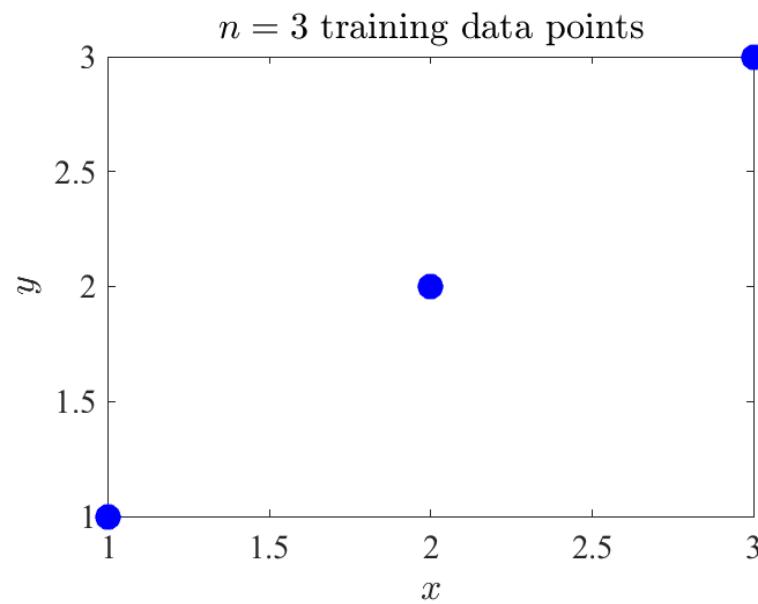
Training a model means setting its parameters so that the model best fits the data. For this purpose, we first need a measure of how well (or poorly) the model fits the training data

Cost function (Least squares): $J(\theta) = \frac{1}{2} \sum_{i=0}^n (h(x^{(i)}) - y^{(i)})^2$ $\min_{\theta} J(\theta)$

How to choose model parameters?

Cost function (Least squares): $J(\theta) = \frac{1}{2} \sum_{i=0}^n (h(x^{(i)}) - y^{(i)})^2$

Goal: Find the value of θ that leads to the minimum value of $J(\theta)$



Mathematical optimization is a central part of machine learning

The normal equation

Given: a training dataset $(x^{(i)}, y^{(i)})$ that contains n sample, use dot products

$$\begin{bmatrix} h(x^{(1)}) \\ \vdots \\ h(x^{(n)}) \end{bmatrix} = \begin{bmatrix} \langle x^{(1)}, \theta \rangle \\ \vdots \\ \langle x^{(n)}, \theta \rangle \end{bmatrix} = \begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(n)} \end{bmatrix} \theta = X\theta$$

Thus, we can rewrite the previous cost function as

$$J(\theta) = \frac{1}{2} \|X\theta - y\|^2$$

The partial derivatives are all zero at the minimum value

$$\frac{1}{2} (2X^T \cdot X\theta - 2X^T y) = 0$$

$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

This expression is known as the normal equation solution of the least squares problem

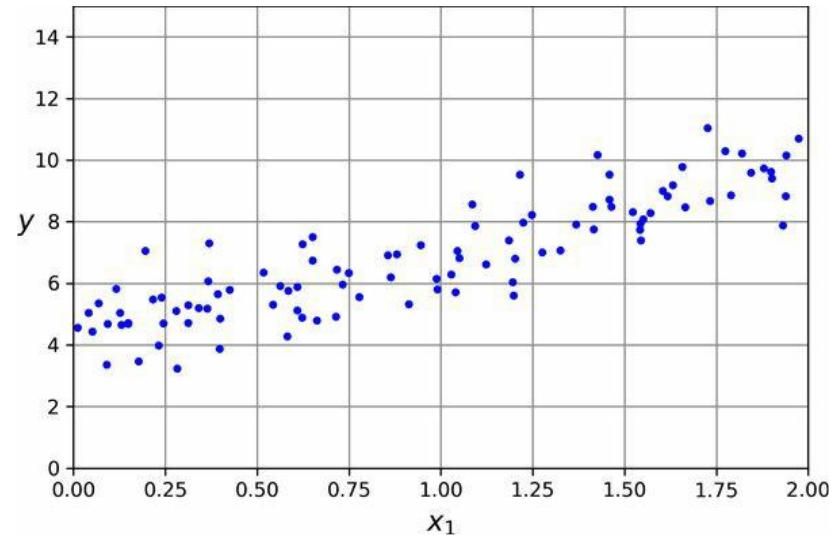
https://en.wikipedia.org/wiki/Least_squares

The normal equation

```
import numpy as np
np.random.seed(42) # to make this code example reproducible
m = 100 # number of instances
X = 2 * np.random.rand(m, 1) # column vector
y = 4 + 3 * X + np.random.randn(m, 1) # column vector

from sklearn.preprocessing import add_dummy_feature
X_b = add_dummy_feature(X) # add  $x_0 = 1$  to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y

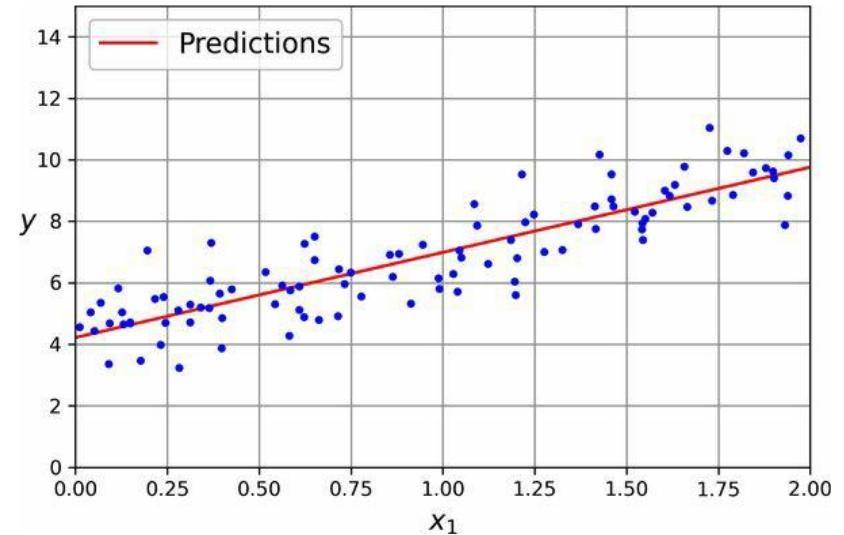
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```



The normal equation

```
>>> X_new = np.array([[0], [2]])  
>>> X_new_b = add_dummy_feature(X_new) # add  $x_0 = 1$  to  
each instance  
>>> y_predict = X_new_b @ theta_best  
>>> y_predict  
array([[4.21509616],  
[9.75532293]])
```

```
import matplotlib.pyplot as plt  
plt.plot(X_new, y_predict, "r-", label="Predictions")  
plt.plot(X, y, "b.")  
[...] # beautify the figure: add labels, axis, grid, and legend  
plt.show()
```



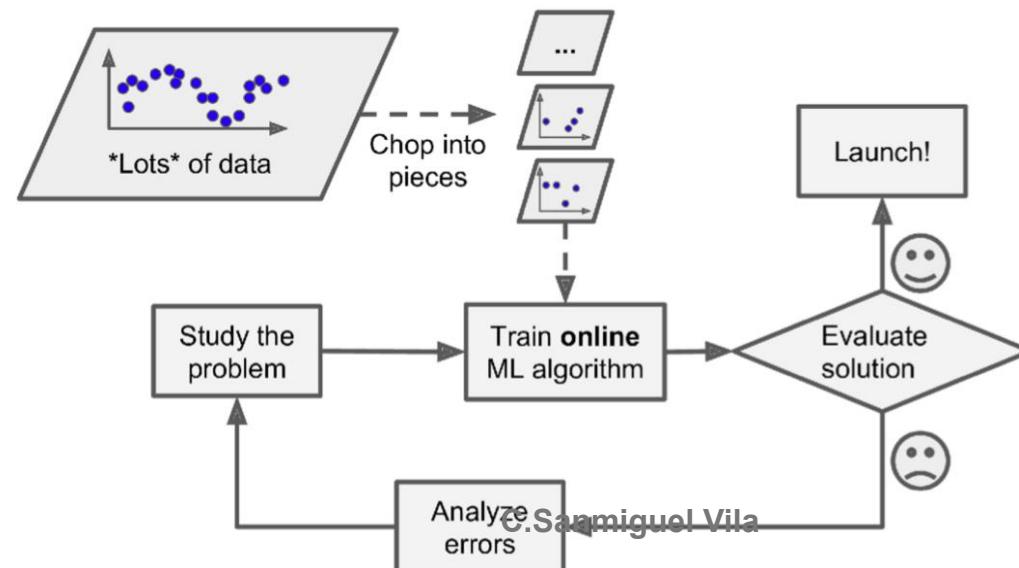
The normal equation

Directly solving it is expensive:

$O(nd^2)$ for the multiplication ($n=\text{samples}$, $d=\text{features}$) and
 $O(d^3)$ for the matrix inverse

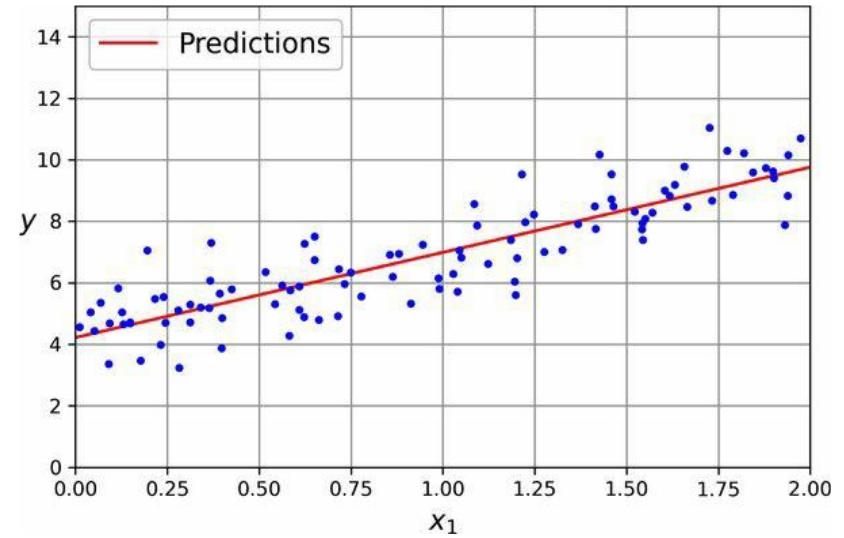
but there are fast methods using SVD or online algorithms.

There are methods (Gradient Descent) that are better suited for cases where there are a large number of features or too many training instances to fit in computer memory.



Scikit-learn

```
>>> from sklearn.linear_model import LinearRegression  
>>> lin_reg = LinearRegression()  
>>> lin_reg.fit(X, y)  
>>> lin_reg.intercept_, lin_reg.coef_  
(array([4.21509616]), array([[2.77011339]]))  
>>> lin_reg.predict(X_new)  
array([[4.21509616],  
[9.75532293]])
```



Notice that Scikit-Learn separates the bias term (`intercept_`) from the feature weights (`coef_`).

Gradient descent

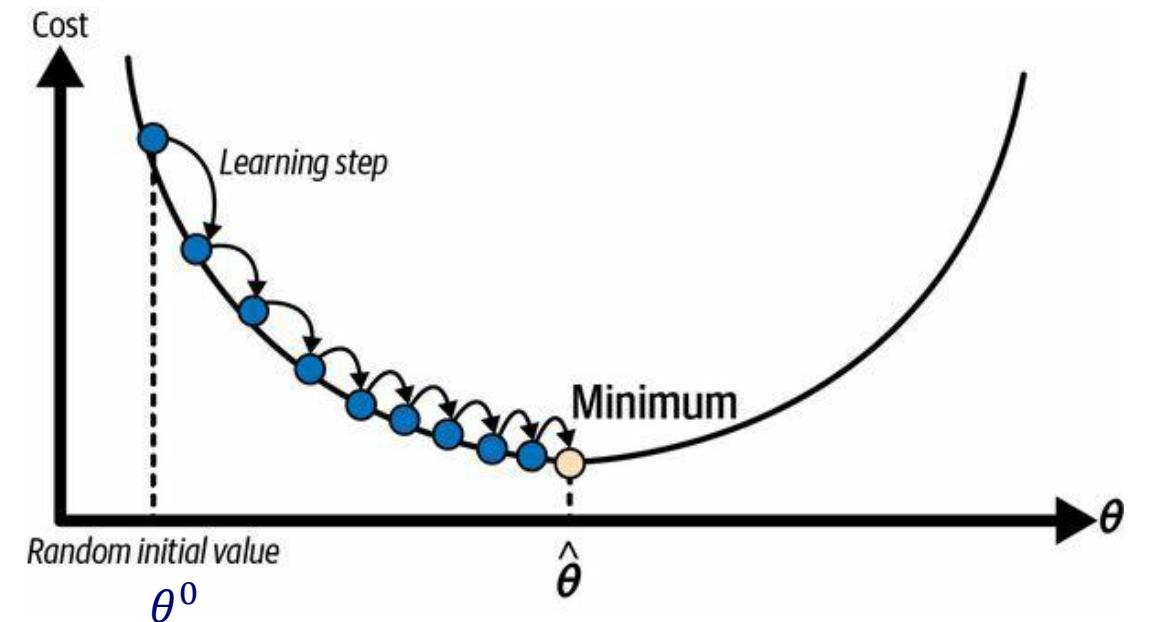
Gradient descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of gradient descent is to tweak parameters iteratively to minimize a cost function

$$\hat{y} = h(x) = \theta x \quad J(\theta) = \frac{1}{2} \sum_{i=0}^n (h(x^{(i)}) - y^{(i)})^2$$

- Start by filling θ with random values
- Take steps to decrease the cost function until convergence

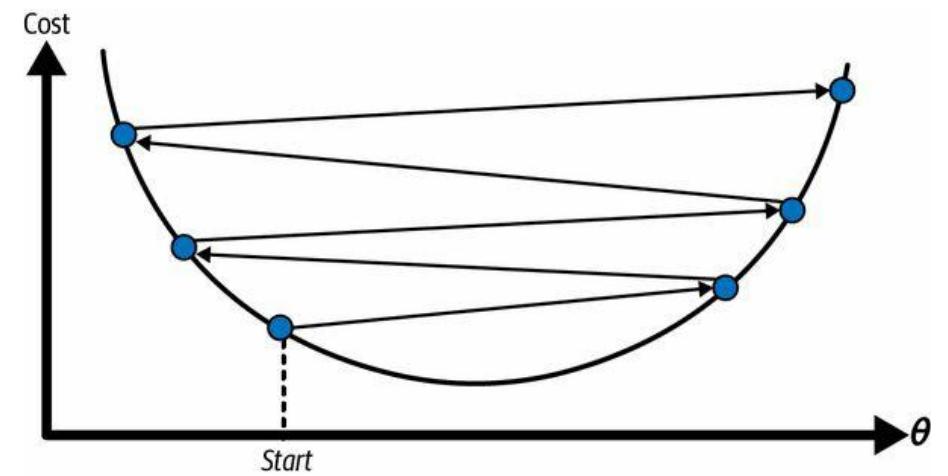
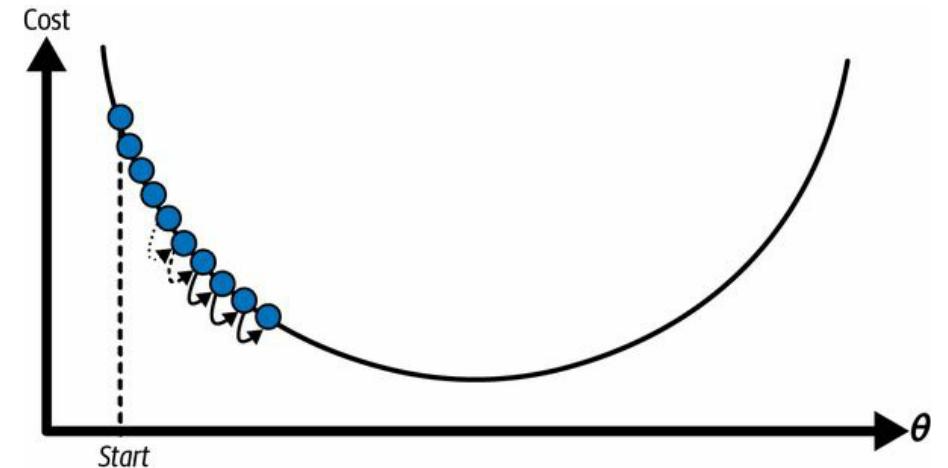
$$\theta^{t+1} = \theta^t - \eta \nabla J(\theta^t)$$

θ^0 is a random initial value
 η the learning rate

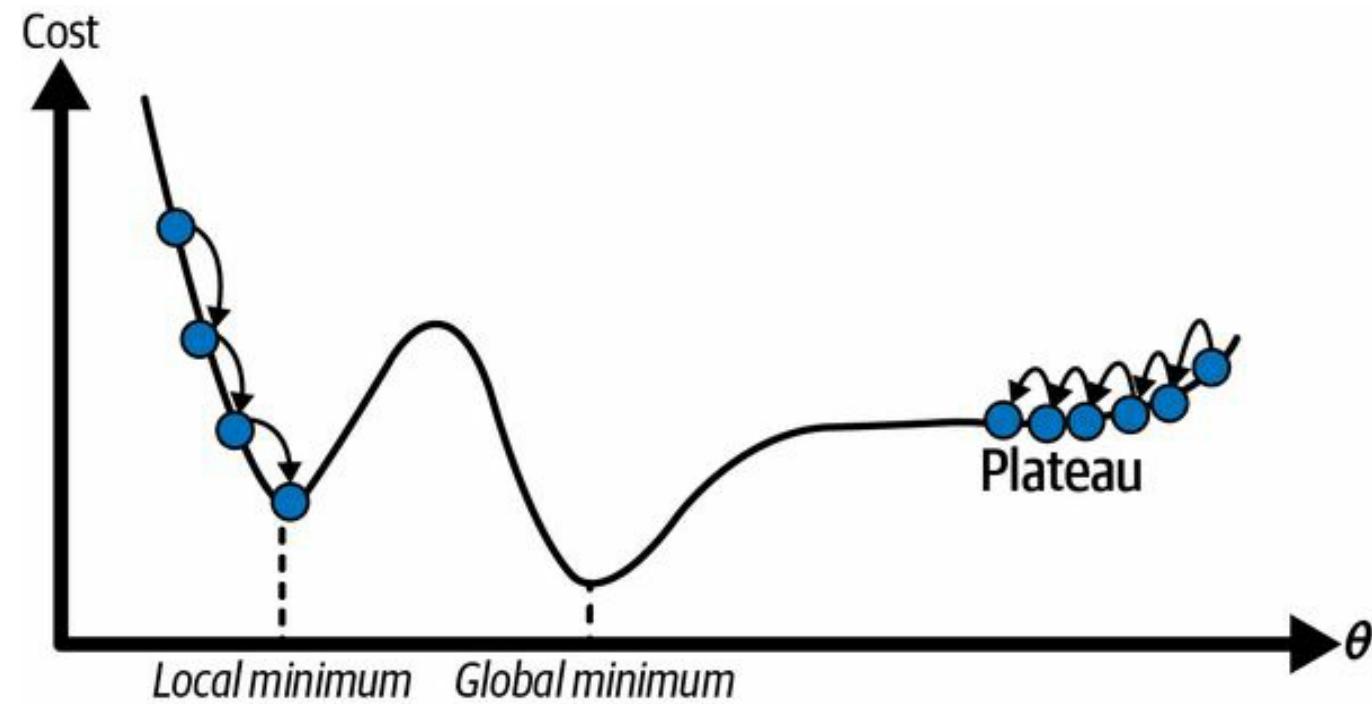


Gradient descent

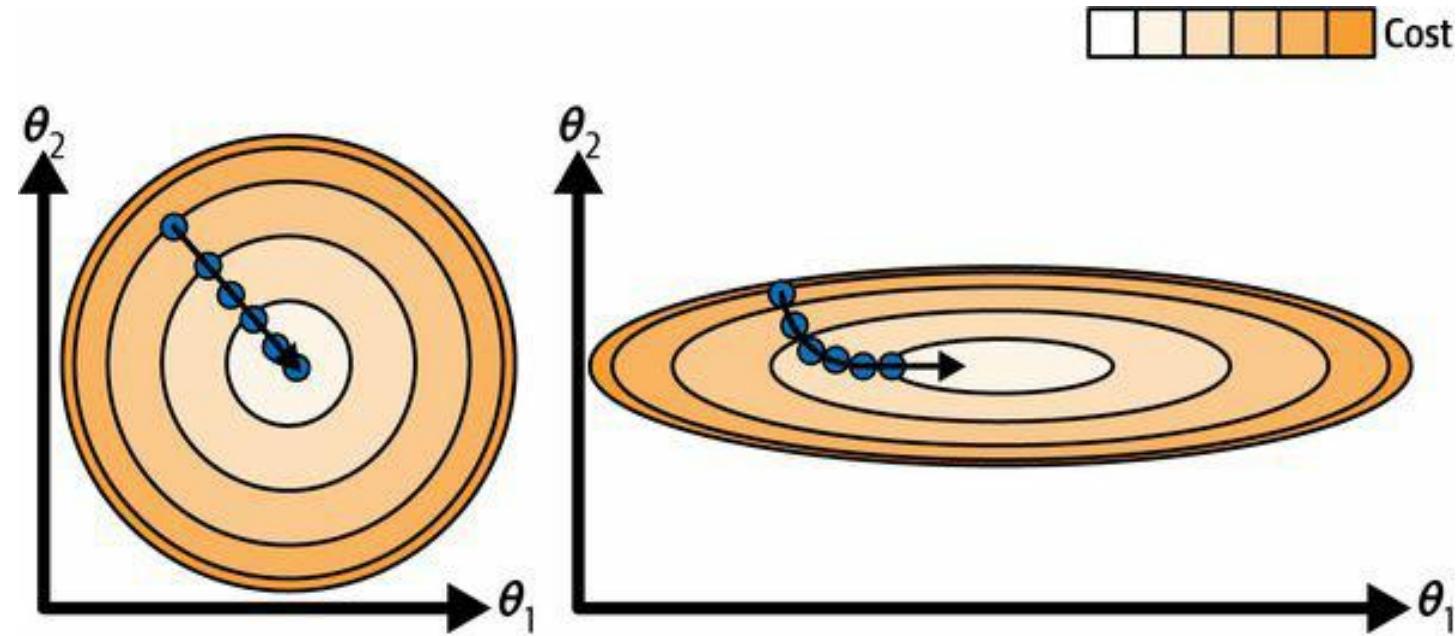
- If the learning rate is too low: many iterations and a long computational time
- If the learning rate is too large: fail to find a good solution



Challenges of using gradient descent



Relevance of feature scaling



Gradient descent with (left) and without (right) feature scaling

When using gradient descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

Gradient Descent Implementation

```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances
np.random.seed(42)
theta = np.random.randn(2, 1) # randomly initialized model
parameters
for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients

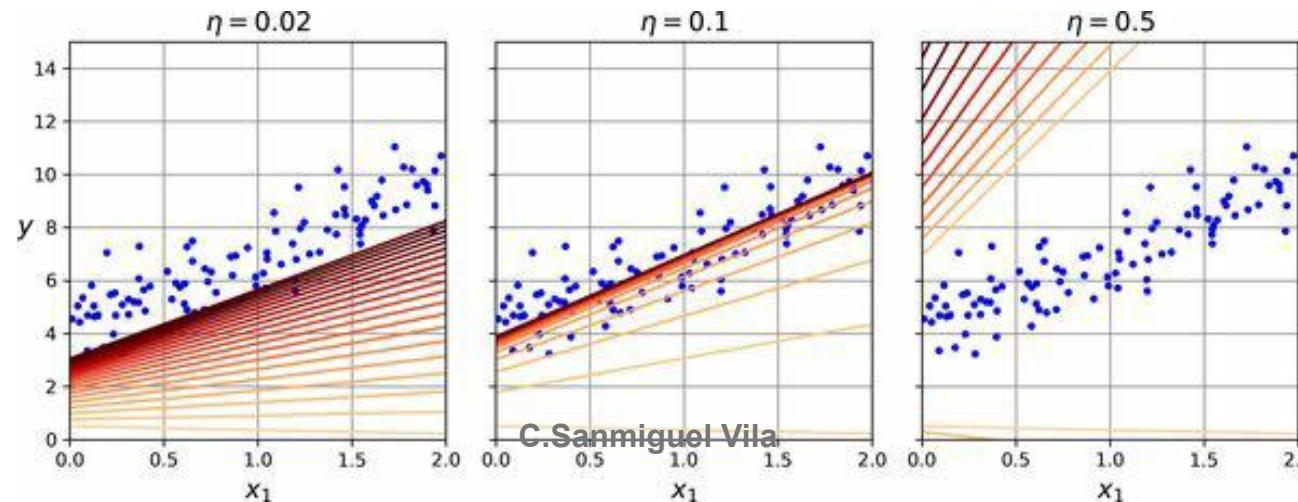
>>> theta
array([[4.21509616],
       [2.77011339]])
```

$$\theta^{t+1} = \theta^t - \eta \nabla J(\theta^t)$$

Gradient Descent Implementation

$$\theta^{t+1} = \theta^t - \eta \nabla J(\theta^t)$$

- This formula involves calculations over the full training set X , at each gradient descent step. This is why the algorithm is called batch gradient descent: it uses the whole batch of training data at every step.
- As a result, it is terribly slow on very large training sets.
- However, gradient descent scales well with the number of features; training a linear regression model with hundreds of thousands of features is much faster using gradient descent than using the Normal equation or SVD decomposition.



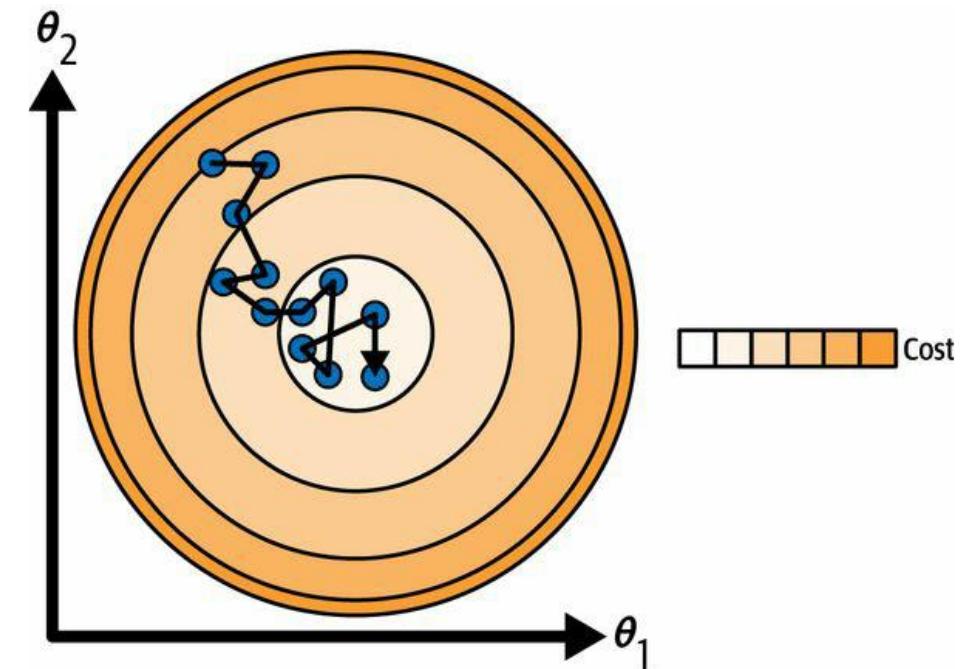
Batch vs Stochastic Minibatch

Batch gradient descent: $\theta = \theta^t - \eta \sum_{i=0}^n ((h_\theta(x^i) - y^i)x^i)$

Stochastic gradient descent picks a random instance in the training set at every step and computes the gradients based only on that single instance.

$$\theta = \theta^t - \eta((h_\theta(x^i) - y^i)x^i)$$

- It is much faster than Batch SD because it has very little data to manipulate at every iteration.
- On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch SD; the cost function will bounce up and down, decreasing only on average.



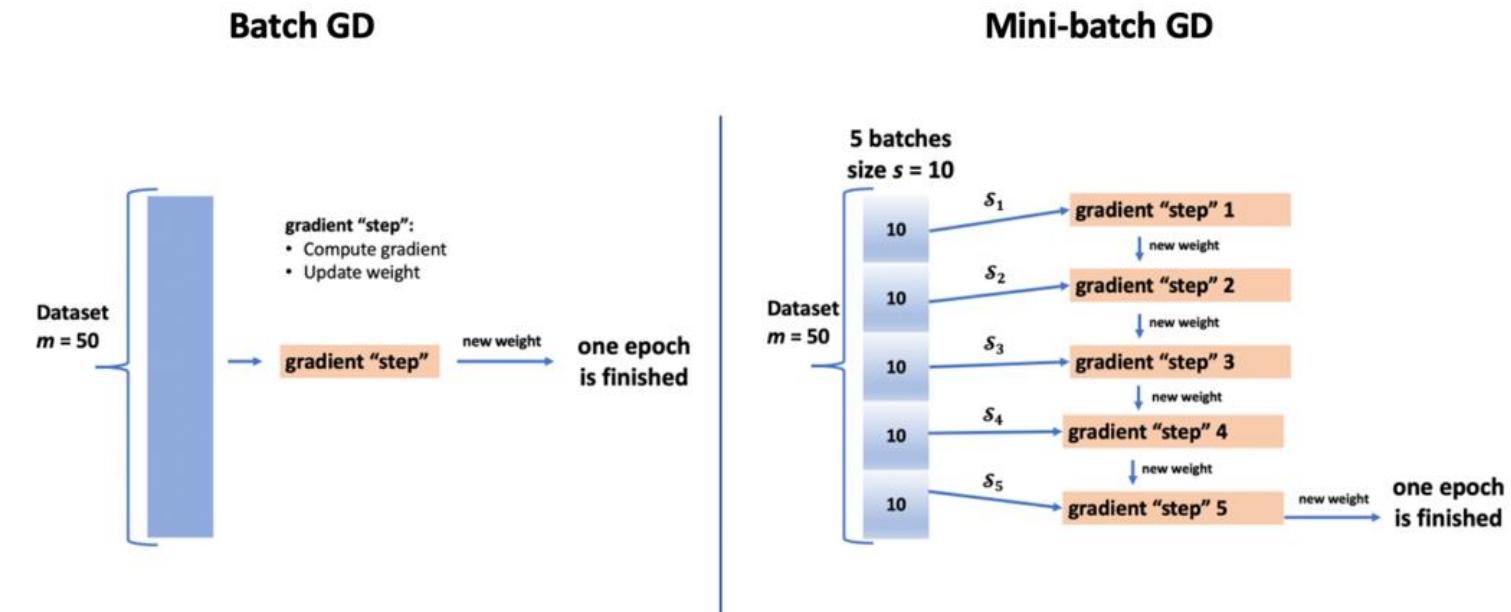
Batch vs Minibatch GD

Batch gradient descent: $\theta = \theta^t - \eta \sum_{i=0}^n ((h_\theta(x^i) - y^i)x^i)$

Minibatch

We take a batch of data (at random) B where $B \ll N$

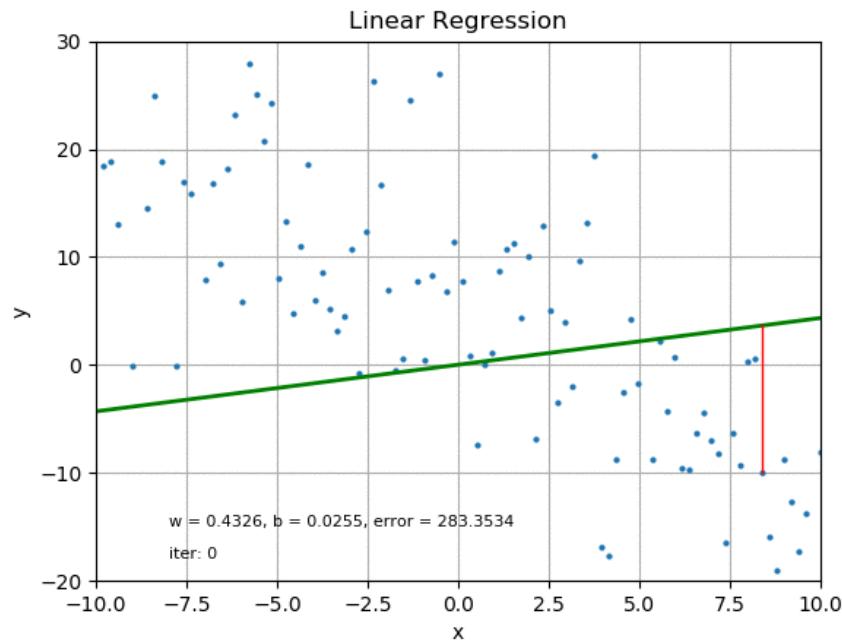
$$\theta = \theta^t - \eta \sum_{i \in B} ((h_\theta(x^i) - y^i)x^i)$$



Batch vs Minibatch GD

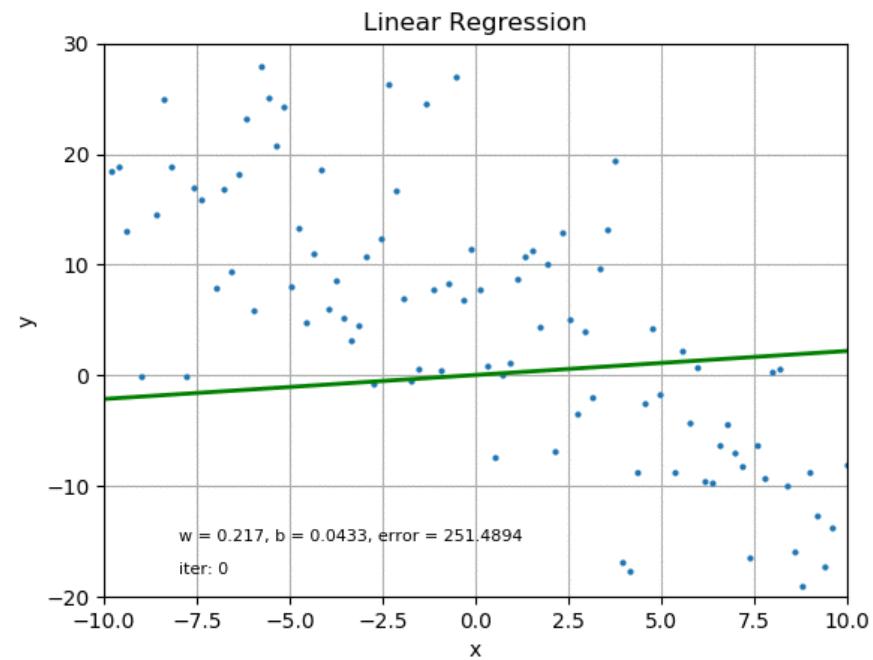
- **Efficient Memory Usage:** Mini-batches allow for better memory utilization, enabling training on large datasets even with limited memory by processing data in smaller chunks.
- **Faster Convergence:** Mini-batch gradient descent often converges faster due to more frequent parameter updates, leading to quicker adjustments towards the optimal solution.
- **Improved Generalization:** The noisy gradient estimates from mini-batches help the model escape local minima and saddle points, resulting in better generalization and robustness.
- **Stochasticity:** Introducing stochasticity through mini-batches helps the model explore the solution space more effectively, reducing the risk of getting stuck in local minima.

Optimization through gradient descent



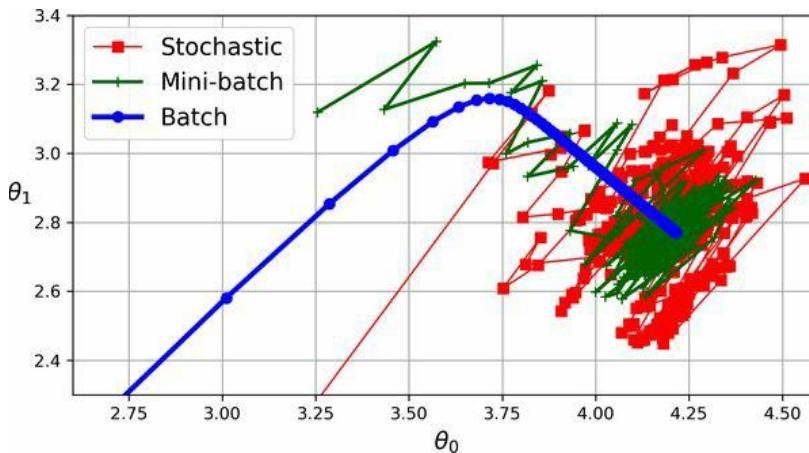
Stochastic gradient descent for
2D linear regression

<https://medium.com/swlh/from-animation-to-intuition-linear-regression-and-logistic-regression-f641a31e1caf>



Batch gradient descent

Comparison of linear regression implementations



Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	SGDRegressor
Stochastic GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor
Mini-batch GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor

Textbook notation: m (number of samples) and n (number of features)

Polynomial regression

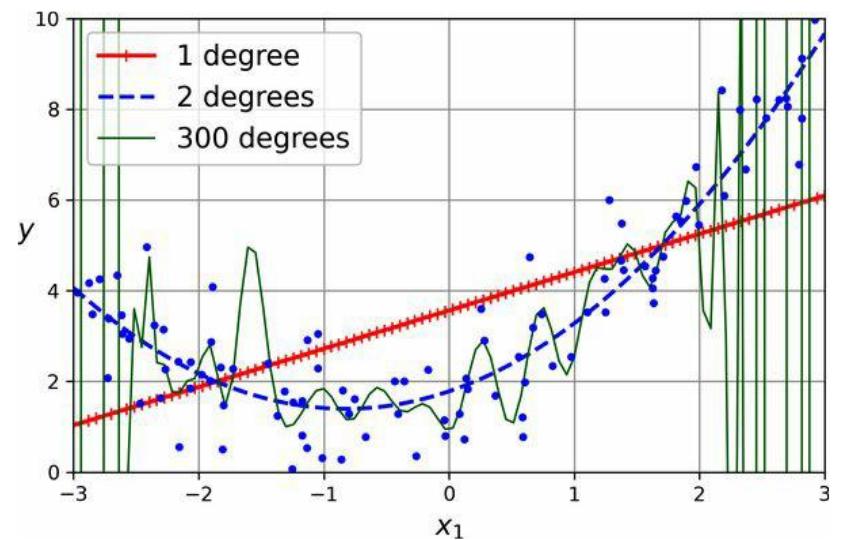
A polynomial of degree 1 gives us the linear regression model

$$h(x) = \theta_0 + \theta_1 x$$

We can add more powers of x as new features

$$h(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \dots$$

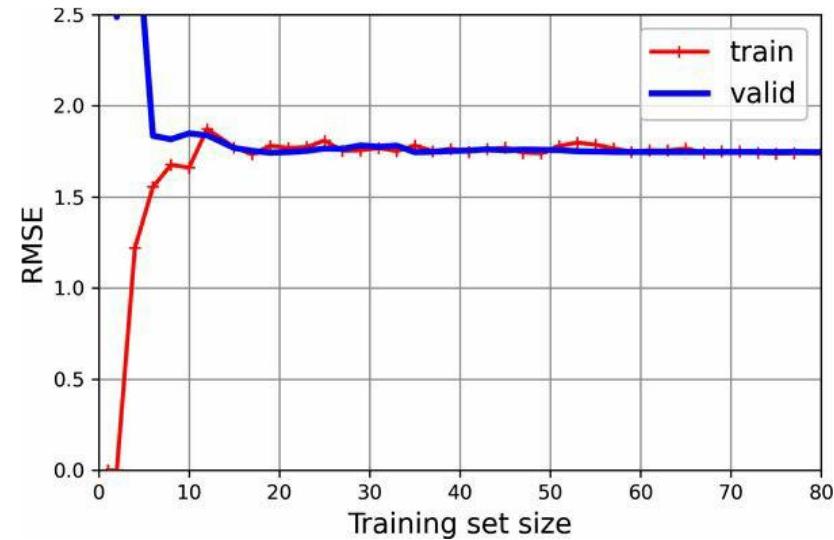
- 300-degree polynomial model wiggles around to get as close as possible to the training instances
- **Overfitting**: performs well on training data but generalizes poorly



Learning curve

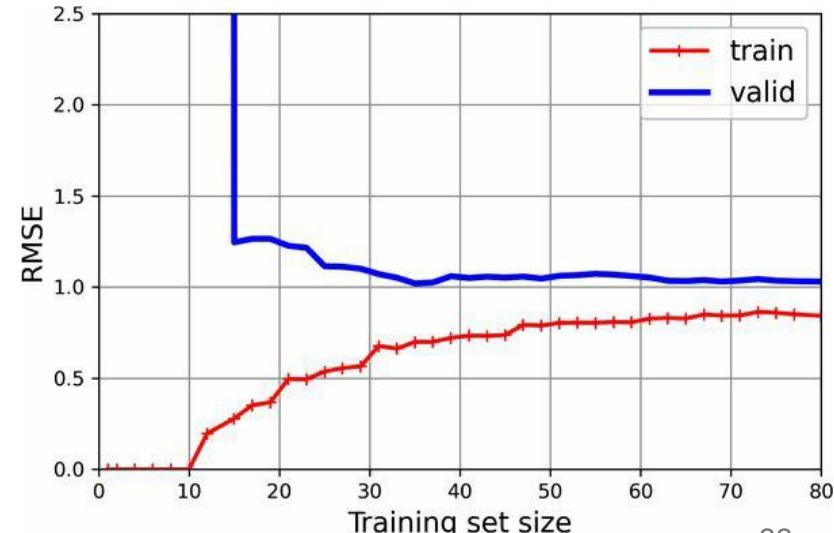
Linear regression

Underfitting: As new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau.



10th-degree polynomial model

- The error on the training data is much lower.
- The gap between the curves means that the model performs better on the training data than on the validation data, which is the hallmark of an overfitting model.
- Using a larger training set, however, the two curves would continue to get closer.



Bias/variance tradeoff

- Generalization error can be expressed as the sum of three different errors:
 1. **Bias:** This is due to wrong assumptions, such as assuming the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data
 2. **Variance:** This is due to the model's excessive sensitivity to small variations in the training data (i.e., degrees of freedom). These models are likely to have high variance and thus overfit the training data.
 3. **Irreducible error:** This is due to the noisiness of the data. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a trade-off.

Ridge (L2) Regression

We can adapt the linear regression to incorporate regularization to prevent overfitting since it tries to prevent a model from becoming too complex.

The cost function is modified to include a penalty term (also called the regularization term) based on the sum of squared weights.

$$\text{Cost function: } J(\theta) = \frac{1}{2} \sum_{i=0}^n (\theta x^{(i)} - y^{(i)})^2 + \alpha \sum_{j=0}^{j=d} \theta_j^2 \quad n=\text{samples}, d=\text{features}$$

Where α is the regularization parameter (also called the penalty term). $\alpha \sum_{j=0}^{j=d} \theta_j^2$ is the L2 regularization term, penalizing large θ_j .

α shrinks the coefficients, reducing their magnitude, which controls overfitting.

$\alpha = 0$ No regularization (equivalent to ordinary least squares regression).

$\alpha = \infty$ Strong regularization, potentially leading to coefficients being close to zero (though not exactly zero).

Ridge (L2) Regression

Advantages:

- Prevents overfitting by introducing bias to the model.
- Handles multicollinearity well by stabilizing estimates.

Disadvantages:

- Shrinks all coefficients uniformly, even the relevant ones.
- Cannot perform feature selection (i.e., it doesn't zero out coefficients).

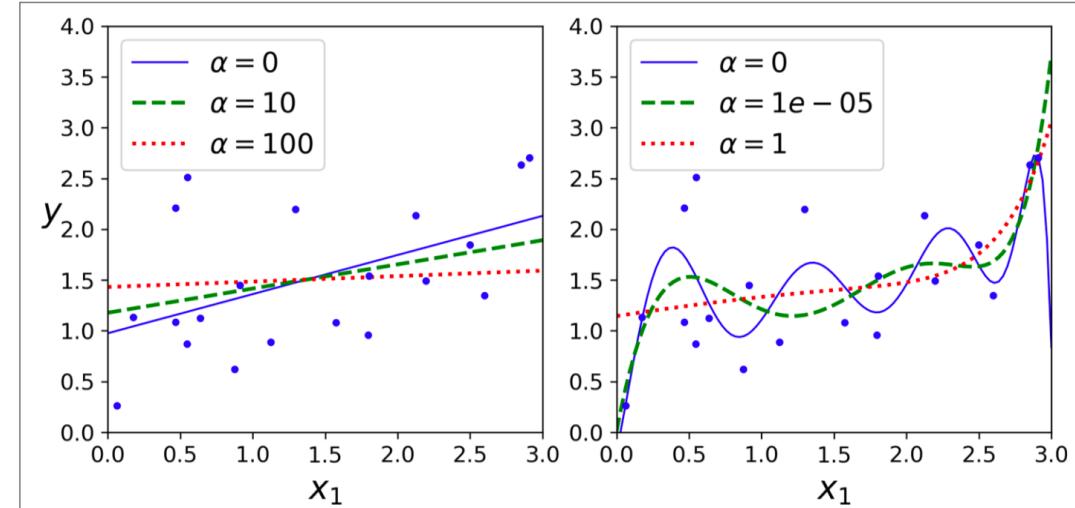


Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

Multicollinearity: some features in your model are **highly correlated**. For least squares (LS) regression, it is complex to estimate the coefficients reliably, as small changes in the data can lead to large fluctuations in the estimates.

Example: Assuming two highly correlated features, x_1 and x_2 . In LS regression, small changes in the data might lead to very different coefficients for θ_1 and θ_2 , making the model unstable. Ridge shrinks both θ_1 and θ_2 , ensuring that the model does not rely too heavily on either, leading to more reliable predictions.

This ability to **stabilize estimates in the face of multicollinearity** is why Ridge Regression is often preferred when working with highly correlated data.

Encouraging “Sparse” Models: Lasso (L1)

Lasso Regression is another type of linear regression that includes regularization, but unlike Ridge, it uses an L1 penalty. This type of regression is useful for feature selection because it can shrink some coefficients to exactly zero, effectively removing irrelevant features.

Cost function: $J(\theta) = \frac{1}{2} \sum_{i=0}^n (x^{(i)}\theta - y^{(i)})^2 + \alpha \sum_{j=0}^{j=d} |\theta_j|$ $n=samples, d=features$

Where α is the regularization parameter (also called the penalty term). $\alpha \sum_{j=0}^{j=d} |\theta_j|$ is the L1 regularization term, penalizing large absolute values of θ_j .

- $\alpha = 0$ No regularization (equivalent to ordinary least squares regression).
- $\alpha = \infty$: Strong regularization, where more coefficients shrink to zero.

Encouraging “Sparse” Models: Lasso (L1)

Advantages:

- **Feature selection:** Lasso can set some coefficients to zero, effectively excluding them from the model.
- Helps in dealing with **overfitting** when there are many irrelevant features.

Disadvantages:

- Can struggle with **highly correlated features**, arbitrarily selecting one and shrinking the rest.
- It might underperform when all features are important but only need some shrinkage.

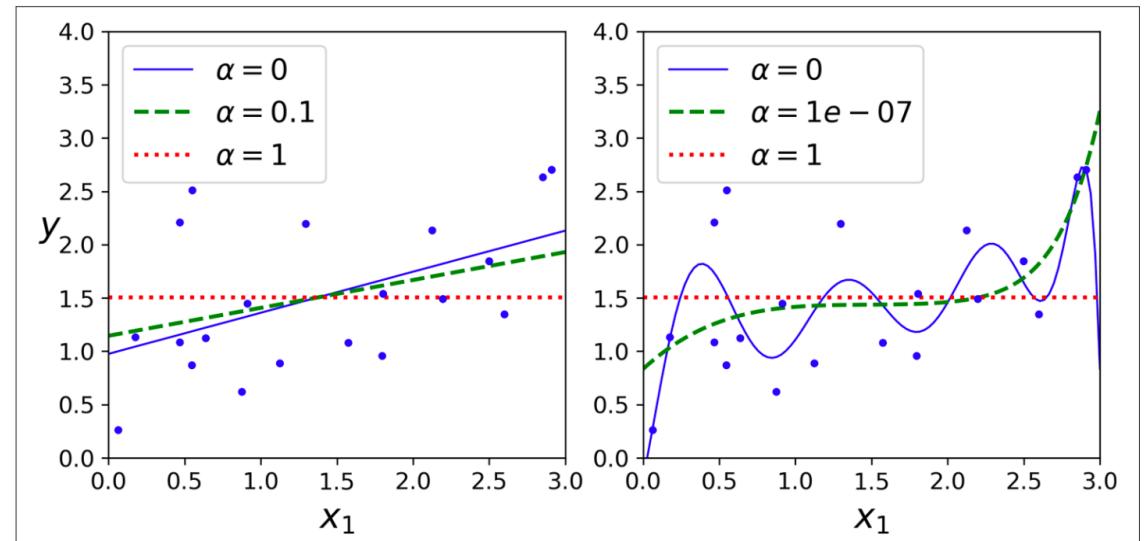


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

Lasso vs Ridge

Advantages:

- **Feature selection:** Lasso can set some coefficients to zero, effectively excluding them from the model.
- Helps in dealing with **overfitting** when there are many irrelevant features.

Disadvantages:

- Can struggle with **highly correlated features**, arbitrarily selecting one and shrinking the rest.
- It might underperform when all features are important but only need some shrinkage.

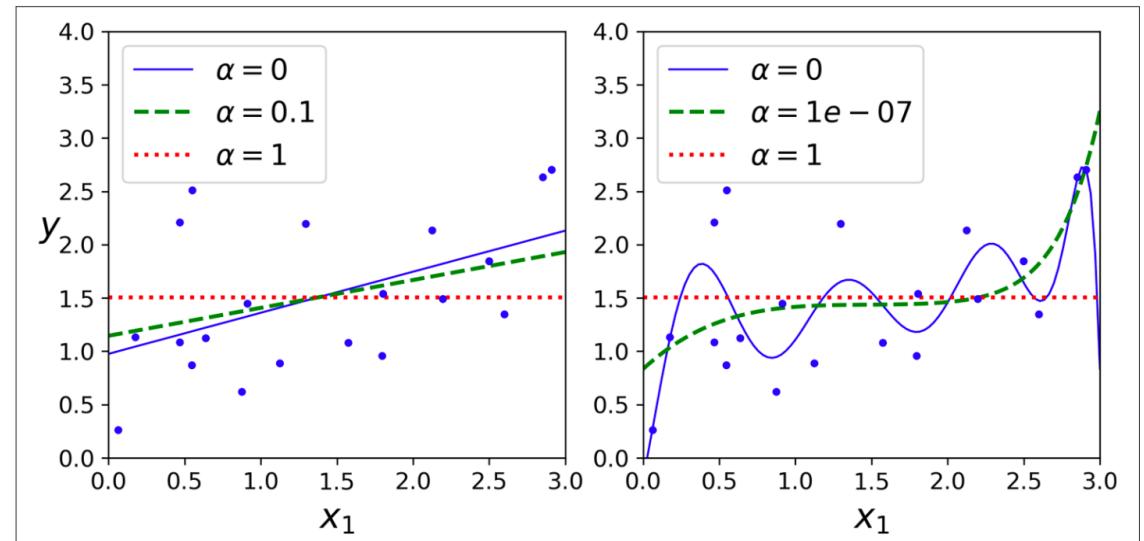


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

Lasso vs Ridge

- **Ridge:** Useful when you have many small/medium-sized coefficients and when multicollinearity is an issue.
- **Lasso:** Useful when you want feature selection and believe many features are irrelevant.
- Can we put the two together?

Elastic Net Regression:L1 + L2

Elastic Net combines the **L1 penalty** from Lasso and the **L2 penalty** from Ridge to benefit from both:

$$\text{Cost function: } J(\theta) = \frac{1}{2} \sum_{i=0}^n (\theta x^{(i)} - y^{(i)})^2 + \alpha_1 \sum_{j=0}^{j=d} \theta_j^2 + \alpha_2 \sum_{j=0}^{j=d} |\theta_j|$$

Why Elastic Net? Solves the limitations of Lasso in high-dimensional datasets where multicollinearity is present.

Balances feature selection (L1) with coefficient shrinkage (L2).

Should we always use Elastic Net?

1. Increased Complexity:

1. Elastic Net introduces **two hyperparameters** that need to be tuned. This adds complexity to the model compared to using only Ridge or Lasso, where there is only one regularization parameter.

2. Risk of Over-regularization:

1. If the **penalty terms are not chosen carefully**, there's a risk of over-regularization, where useful features might be shrunk too much, leading to underfitting (where the model is too simple and fails to capture important relationships).
2. It requires a balance between L1 and L2 penalties, which can be tricky to optimize.

3. Interpretability:

1. Elastic Net might keep more features in the model than Lasso, making it harder to interpret when feature selection is important..

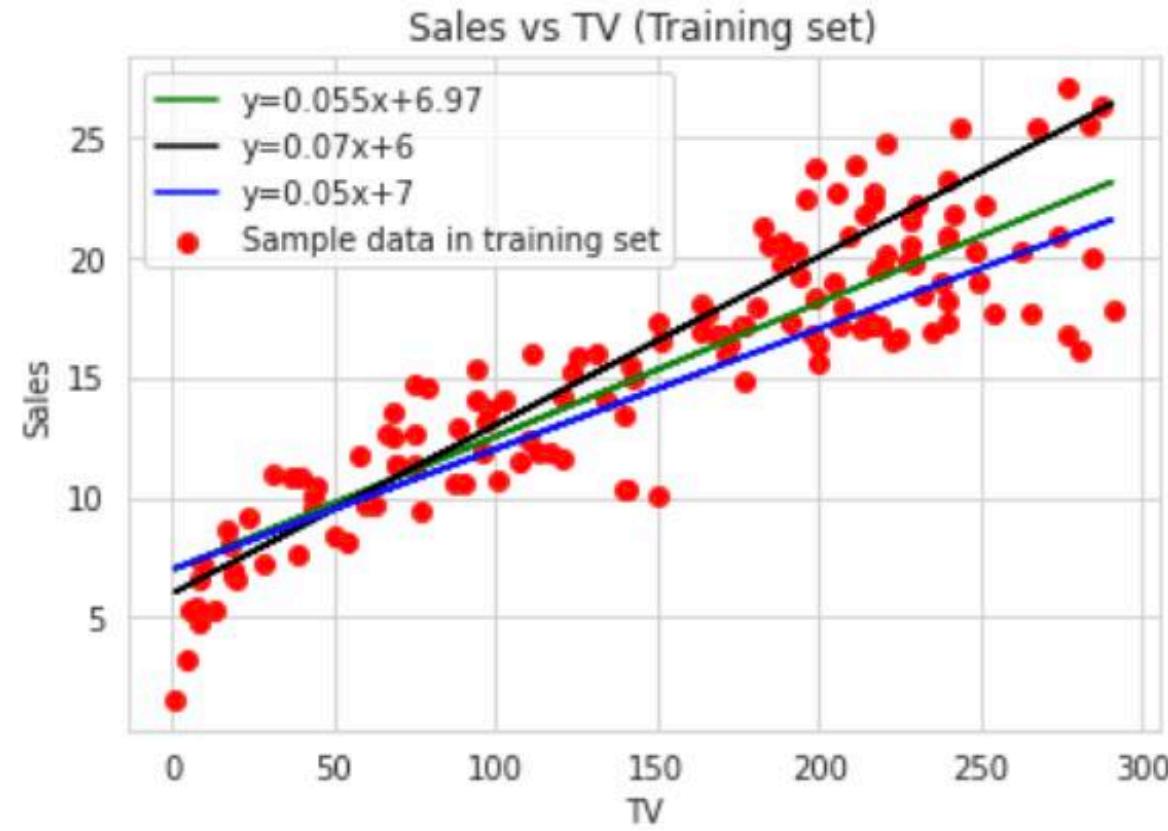
4. Lasso Limitations with Multicollinearity:

1. It might still **struggle with strongly correlated features**. Ridge helps distribute weight across correlated variables, but Elastic Net's L1 component can still arbitrarily pick one variable over another in highly collinear scenarios.

5. Requires Sufficient Data:

1. Elastic Net is most effective when there are more features than observations or when some features are correlated. However, in smaller datasets, the dual regularization may not provide significant advantages, and tuning can be difficult due to limited data.

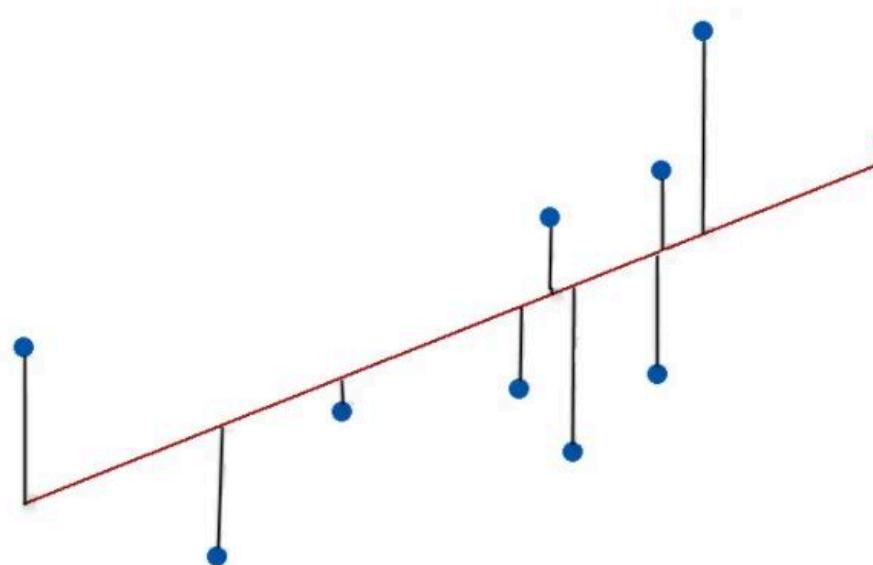
Regression Metrics



Which model is better?

Regression Metrics

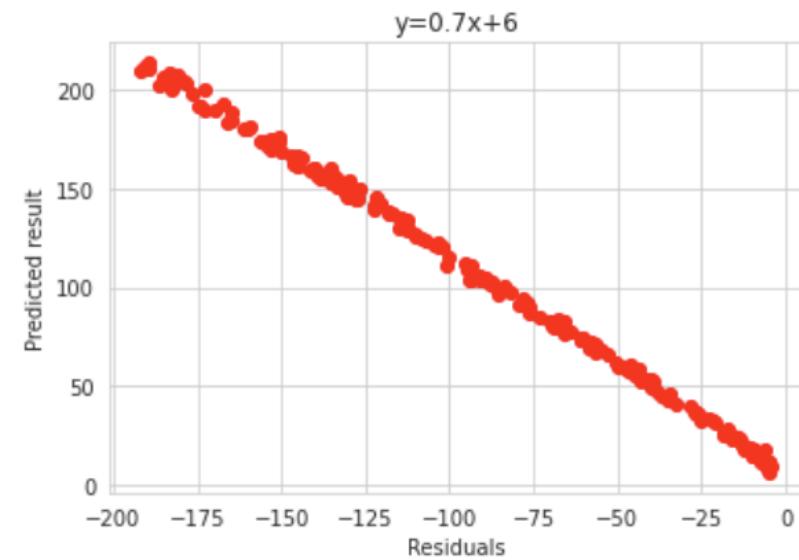
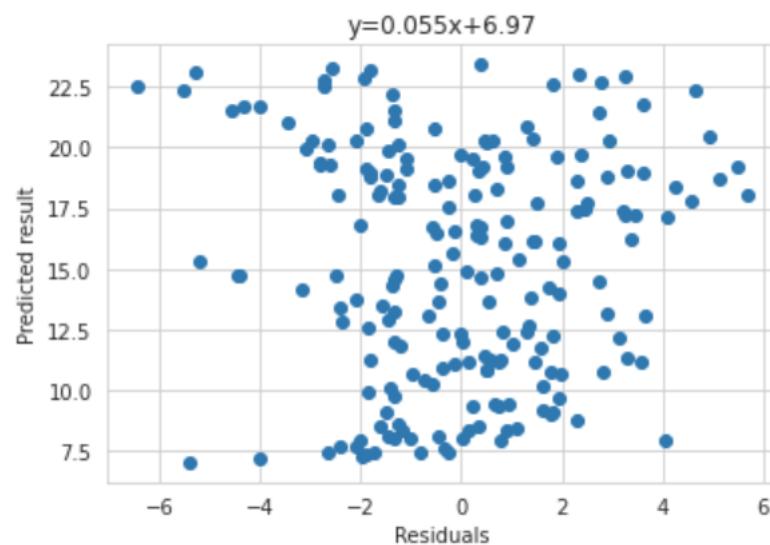
- Residuals refer to the difference between actual values and their predicted values



- We want a regression model where residuals are small and unbiased (i.e., random)

Plot the residuals

Which one is better?



You can't trust a model that is biased

R-squared (R^2) value

- R^2 is always between 0 and 1
 - 0: means a model does not explain any variation in the target variable around its mean (not desirable)
 - 1: means a model that explains all the variation in the target variable around its mean (desirable)

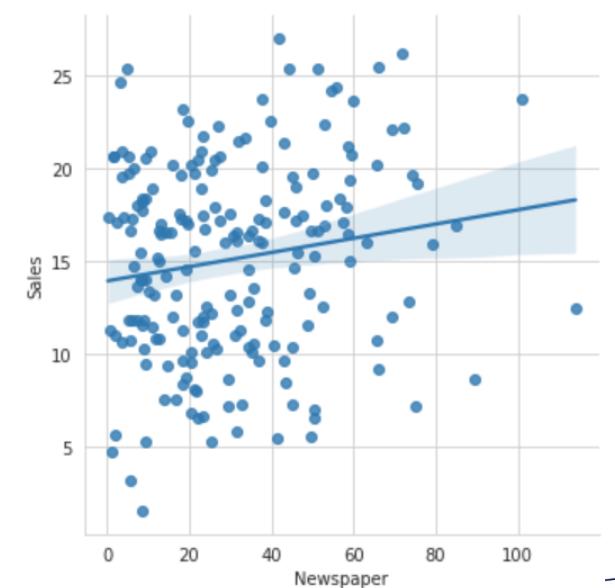
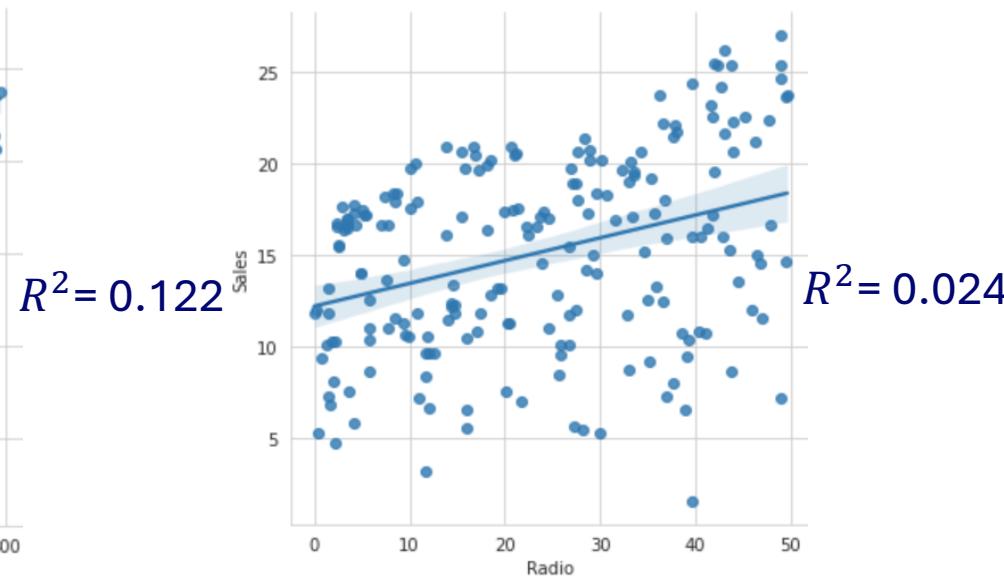
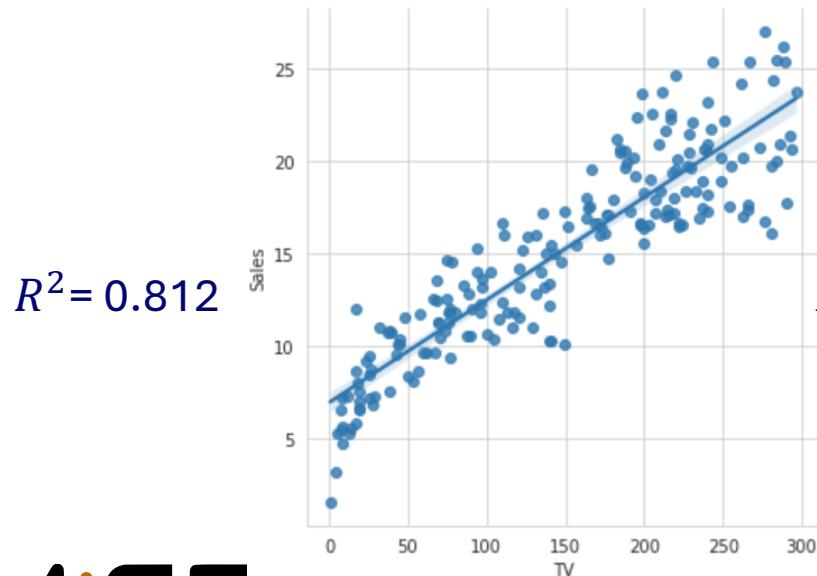
$$R^2 = 1 - \frac{RSS}{TSS}$$

$RSS = \sum_i (y^{(i)} - \hat{y}^{(i)})^2$: sum of squares of residuals

$TSS = \sum_i (y^{(i)} - \bar{y})^2$: total sum of squares (i.e., variance)

R-squared (R^2) value

- R^2 evaluates the scatter of the data points around the regression line. It is also called as coefficient of determination
- The higher R^2 is, the smaller difference between the observed data and the predicted values



Performance Metrics

- We want to evaluate our regression model on the testing set
- Possible Metrics:
 - Mean Squared Error (MSE)
 - Root Mean Squared Error (RMSE)
 - Mean Absolute Error (MAE)

MSE & RMSE

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

$$RMSE = \sqrt{MSE}$$

i	1	2	3	4	5	6	7
Predicted	2.5	0	2	8	1	1	-6
True	3	-0.5	2	7	2	2	-5

$$MSE = \frac{1}{7} \sum_{i=1}^7 (3 - 2.5)^2 + (-0.5 - 0)^2 + (2 - 2)^2 + (7 - 8)^2 + (2 - 1)^2 + (2 - 1)^2 + (-5 - (-6))^2$$

```
1 from matplotlib import pyplot
2 from sklearn.metrics import mean_squared_error
3 # real value
4 y_true = [3, -0.5, 2, 7, 2, 2, -5]
5 # predicted value
6 y_predicted = [2.5, 0, 2, 8, 1, 1, -6]
7 # calculate errors
8 mean_squared_error(y_true, y_predicted)
```

0.6428571428571429

Mean Absolute Error (MAE)

$$MAE = \frac{1}{n} \sum_{i=1}^n |y^{(i)} - \hat{y}^{(i)}|$$

i	1	2	3	4	5	6	7
Predicted	2.5	0	2	8	1	1	-6
True	3	-0.5	2	7	2	2	-5

$$MAE = \frac{1}{7} \sum_{i=1}^7 |3 - 2.5| + |-0.5 - 0| + |2 - 2| + |7 - 8| + |2 - 1| + |2 - 1| + |-5 - (-6)|$$

```
1 from matplotlib import pyplot
2 from sklearn.metrics import mean_absolute_error
3 # real value
4 y_true = [3, -0.5, 2, 7, 2, 2, -5]
5 # predicted value
6 y_predicted = [2.5, 0, 2, 8, 1, 1, -6]
7 # calculate errors
8 mean_absolute_error(y_true, y_predicted)
```

0.7142857142857143

C.Sanmiguel Vila

40



MSE vs RMSE vs MAE

- MSE and RMSE penalize large prediction errors via squared function
 - RMSE is more commonly used than MSE as it has the same unit as Y
- MAE is more robust to outliers
- MSE and RMSE are differentiable functions while MAE is not
 - i.e., its derivative exists for all values
 - This makes RMSE used as a default metric for loss function in most ML models
 - The lower the RMSE/MSE is, the better the model is
 - Scoring function:
`neg_root_mean_squared_error/neg_mean_squared_error`
 - https://scikit-learn.org/stable/modules/model_evaluation.html

Negative RMSE & MAE

- In scikit-learn, the metrics for some scoring functions (like cross-validation) are framed so that maximization is preferred. Since RMSE needs to be minimized (lower is better), scikit-learn returns the negative value of RMSE in functions like `cross_val_score()` or `GridSearchCV()`, where it expects scores to be maximized. This ensures that a higher (less negative) score is better during optimization.

Negative RMSE & MAE

When to use negative RMSE

- You use negative RMSE when scikit-learn expects scores that need to be maximized, such as Cross-validation with `cross_val_score()` or other scikit-learn functions such as hyperparameter tuning with `GridSearchCV()` or `RandomizedSearchCV()`
- In these cases, scikit-learn requires a metric where higher scores indicate better performance (since these methods are designed to maximize scores). RMSE, on the other hand, is a metric where lower values are better, so scikit-learn returns its negative value to make the scoring consistent with its optimization framework (where higher scores are considered better)
- By using negative errors, scikit-learn turns the problem of minimizing RMSE into one where it's maximizing the negative of RMSE. As a result, a less negative score (closer to 0) indicates better performance

Negative RMSE & MAE

When to use RMSE:

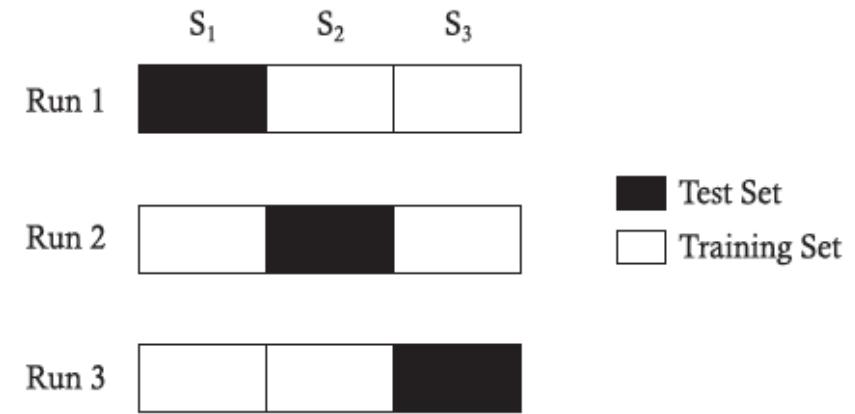
- You use RMSE when manually calculating the RMSE outside of scikit-learn's cross-validation or optimization functions and are interested in the actual RMSE value for interpretation or model comparison.
- For example, you might use RMSE in:
 - Manual evaluation of a model's performance on a test set.
 - Reporting final model performance to give a direct sense of error in the target's units (e.g., predicting house prices or temperature).

Avoid RMSE:

- If you're dealing with data that contains many outliers or non-normally distributed residuals, you might prefer using **MAE** (Mean Absolute Error), which is less sensitive to outliers.

Cross validation for Linear Regression

- **Avoid the performance evaluation being sensitive to the selection of training data**
- **K-fold cross validation:**
 - Divide the whole dataset into K partitions
 - ($K-1$) partitions for training, 1 partition for testing
 - Rotate the selection of the testing set
 - Average the evaluation



Cross validation for Linear Regression

```
1 from sklearn.linear_model import LinearRegression
2 from sklearn.model_selection import cross_val_score
3 import numpy as np
4 lr=LinearRegression()
5 x=df.drop('Sales', axis=1).values
6 y=df['Sales'].values
7 cv_result=cross_val_score(lr, x, y, cv=5, scoring='neg_mean_squared_error')
8 print(cv_result)
9 score=np.mean(cv_result)
10 print(score)
```

```
[-3.05606897 -2.02676065 -1.85105212 -4.72039259 -2.63694072]
-2.858243009991011
```

Cross validation for Linear Regression

```
1 from sklearn.linear_model import LinearRegression  
2 from sklearn.model_selection import cross_val_score  
3 import numpy as np  
4 lr=LinearRegression()  
5 x=df.drop('Sales', axis=1).values  
6 y=df['Sales'].values  
7 cv_result=cross_val_score(lr, x, y, cv=5, scoring='neg_mean_squared_error')  
8 print(cv_result)  
9 score=np.mean(cv_result)  
10 print(score)
```

```
[-3.05606897 -2.02676065 -1.85105212 -4.72039259 -2.63694072]  
-2.858243009991011
```

Data-intensive space engineering

Lecture 3

Carlos Sanmiguel Vila

Logistic regression

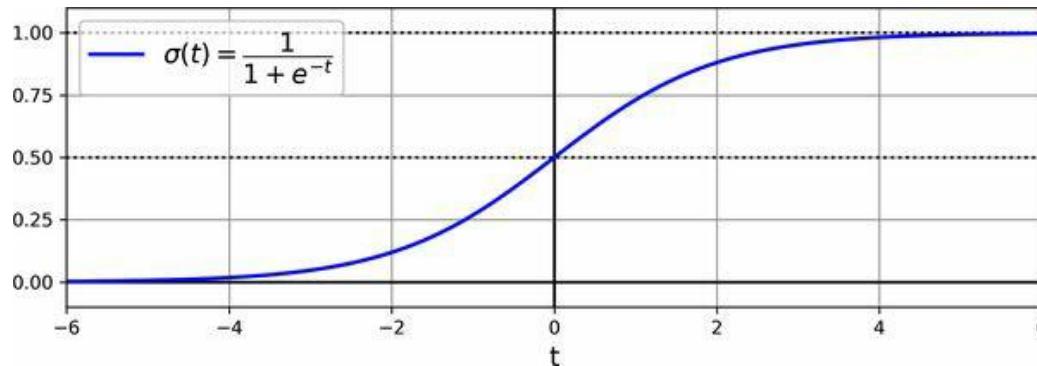
Prediction

- Logistic regression is used to estimate the probability that an instance belongs to a particular class (out of two classes, e.g., an email is spam or not)
- If the estimated probability is greater than 50%, the model predicts that the instance belongs to the positive class (labeled “1”) otherwise, the model predicts that the instance belongs to the negative class (labeled “0”)
- Like linear regression, logistic regression computes a weighted sum of the input features
- **Difference:** It applies the **logistic function** σ to the output result

$$\hat{p} = h(x) = \sigma(\theta_0x_0 + \theta_1x_1 + \cdots + \theta_dx_d)$$

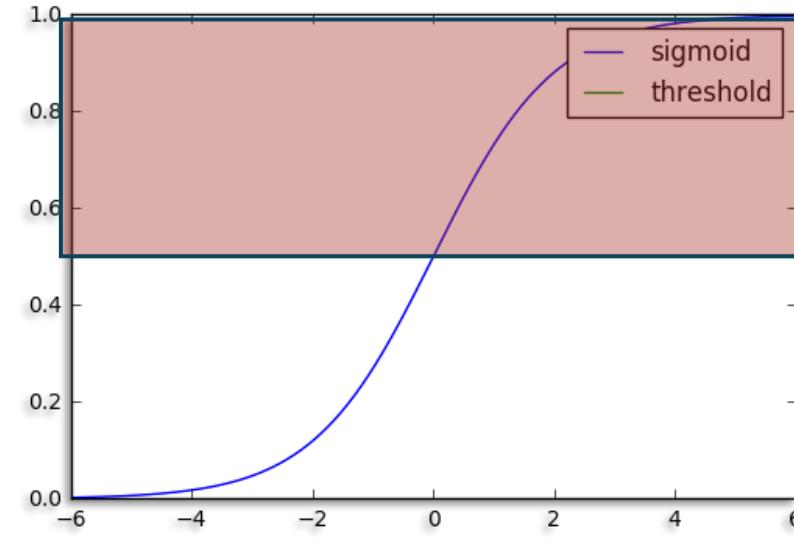
Logistic function

- The logistic or sigmoid function outputs a number between and as follows:



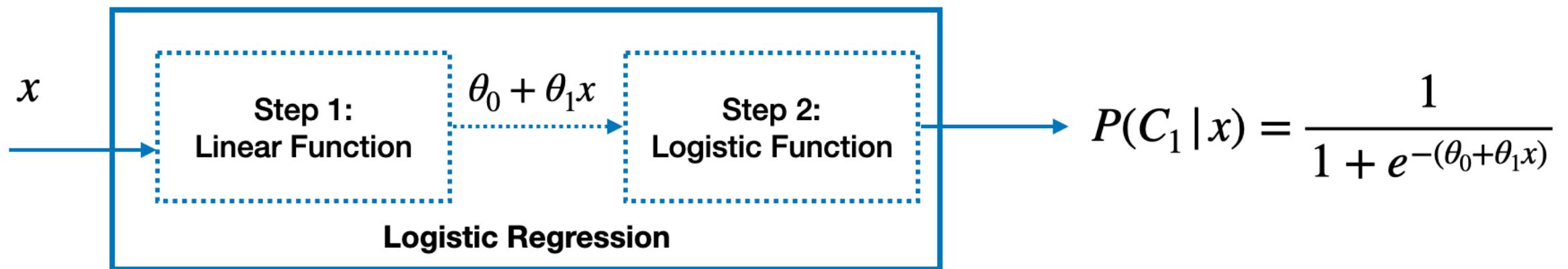
$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

- Using a 50% threshold probability
 $\hat{y} = 0$ if $\hat{p} < 0.5$ and if $\hat{p} > 0.5$ $\hat{y} = 1$



Logistic function

- Assuming a two-class classification problem C_1, C_2



- We want to find $P(C_i | x)$

$$P(C_1 | x) + P(C_2 | x) = 1 \rightarrow P(C_1 | x) = 1 - P(C_2 | x)$$

Decision boundary

Consider, $\hat{p} = h(x) = \sigma(\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2)$

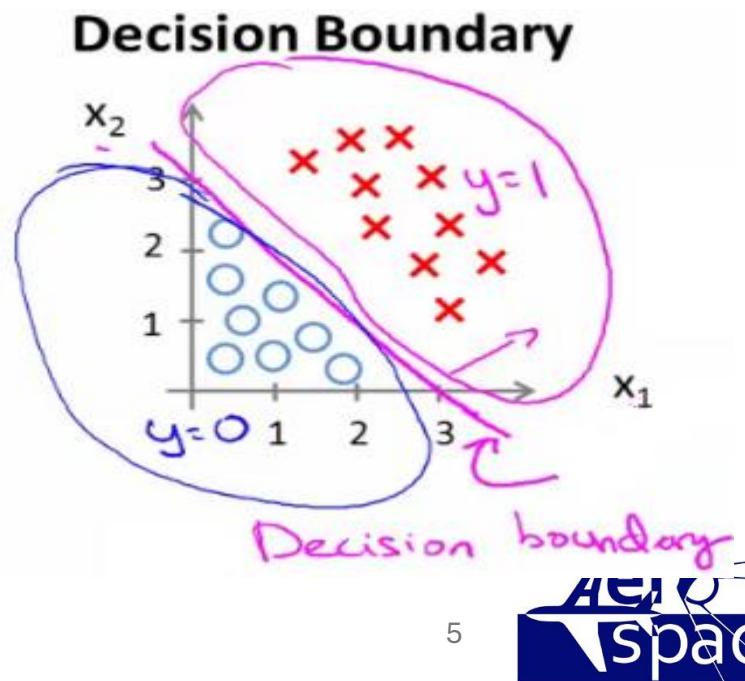
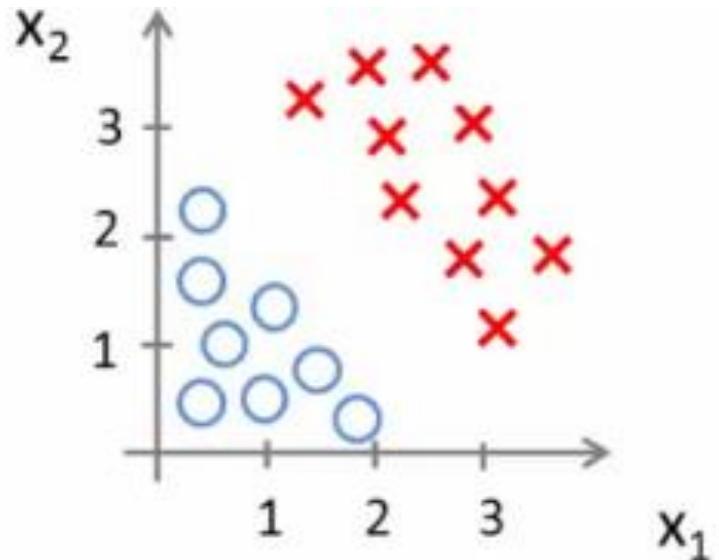
For example, $\theta = [-3, 1, 2]$

- What does this mean?

We predict $\hat{y} = 1$ if $-3x_0 + 1x_1 + 1x_2 \geq 0$

- We can also re-write this as If $(x_1 + x_2 \geq 3)$ then we predict $\hat{y} = 1$

- If we plot $x_1 + x_2 = 3$ we graphically plot our **decision boundary**



Decision boundary

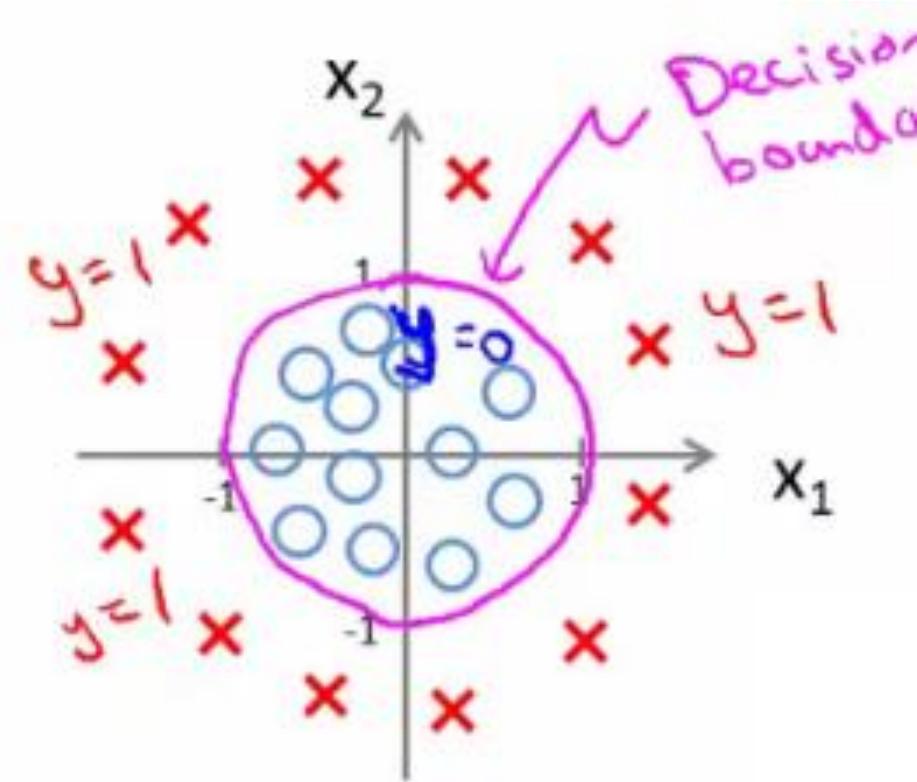
Using non-linear functions,

$$\hat{p} = h(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_3 x_1^2 + \theta_4 x_2^2)$$

For example, $\theta = [-1, 0, 0, 1, 1]$

We predict $\hat{y} = 1$ if:

- $-1 + 1x_1^2 + x_2^2 \geq 0$
- Or $x_1^2 + x_2^2 = 1$



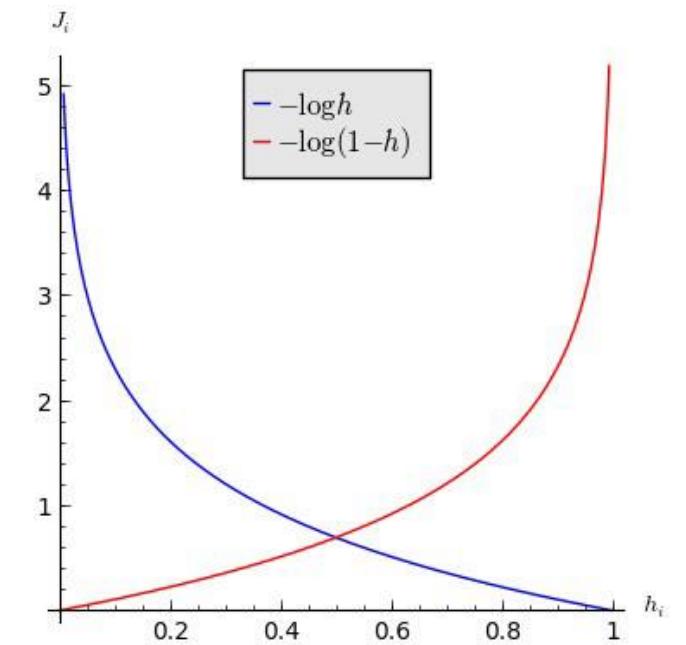
Training and cost function

- To perform the training, we need a cost or loss function other than MSE for logistic regression: MSE with sigmoid is non-convex and gradient descent will get trapped in local minima
- In general terms $J(\theta)$ can be expressed as

$$J(\theta) = \frac{1}{n} \sum_{i=0}^n \text{cost}(h(x^{(i)}), y^{(i)})$$

- For the logistic regression we can use

$$\text{cost}(h(x), y) = \begin{cases} -\log(h(x)) & \text{if } y = 1 \\ -\log(1 - h(x)) & \text{if } y = 0 \end{cases}$$



Training and cost function

- Note that for a two-class problem $y \in \{0,1\}$

$$cost(h(x), y) = -y \log(h(x)) - (1 - y)\log(1 - h(x))$$

- The cost function over the whole training set

$$J(\theta) = -\frac{1}{n} \sum_{i=0}^n [y^{(i)} \log(h(x^i)) + (1 - y^i)\log(1 - h(x^i))]$$

- The bad news is that there is no known closed-form solution, but the good news is that this cost function is convex, and therefore we can use Gradient Descent to find the global minimum

Logistic regression

- The log loss was not just pulled out of a hat. It can be shown mathematically (using Bayesian inference) that minimizing this loss will result in the model with the *maximum likelihood* of being optimal, assuming that the instances follow a Gaussian distribution around the mean of their class. When you use the log loss, this is the implicit assumption you are making. The more wrong this assumption is, the more biased the model will be.
- Similarly, when we used the MSE to train linear regression models, we were implicitly assuming that the data was purely linear, plus some Gaussian noise. So, if the data is not linear (e.g., if it's quadratic) or if the noise is not Gaussian (e.g., if outliers are not exponentially rare), then the model will be biased.

Softmax regression

- The logistic regression model can be generalized to support multiple classes
- Let K be the number of classes and compute a score $s_k(x)$ for each $k \in \{1, \dots, K\}$

$$s_k(x) = \mathbf{x}^T \boldsymbol{\theta}^k$$

- Estimate the probability that the instance x belongs to class k

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

Softmax regression

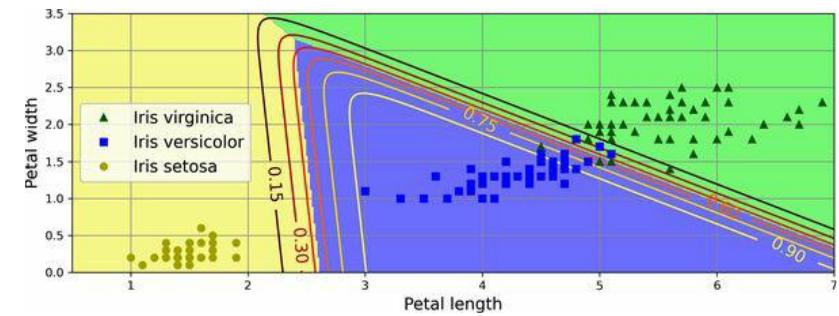
- Softmax regression predicts the class with the highest estimated probability

$$\hat{p} = \operatorname{argmax}_k \sigma(s_k(x))_k = \operatorname{argmax}_k s_k(x) = \operatorname{argmax}_k x^T \theta^k$$

- Cross entropy cost function

$$J(\theta) = -\frac{1}{n} \sum_{i=0}^n \sum_{j=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

- where $y_k^{(i)} \in \{0,1\}$ is the target probability



Confusion Matrix

- Confusion matrix:
 - Defines the performance of a classification model
 - Represented as a table

		PREDICTED CLASS	
		Class=Yes	Class>No
ACTUAL CLASS	Class=Yes	a	b
	Class>No	c	d

a: TP (true positive) b: FN (false negative)

c: FP (false positive) d: TN (true negative)

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
cm_df=pd.DataFrame(cm, index=['Luxury','Economy'], columns=['Luxury','Economy'])
cm_df
```

	Luxury	Economy
Luxury	51	2
Economy	7	58

```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred, normalize='true')
cm_df=pd.DataFrame(cm, index=['Luxury','Economy'], columns=['Luxury','Economy'])
cm_df
```

Normalized on rows

	Luxury	Economy
Luxury	0.962264	0.037736
Economy	0.107692	0.892308

Compute Accuracy from Confusion Matrix

		PREDICTED CLASS	
		Class=Yes	Class>No
ACTUAL CLASS	Class=Yes	a (TP)	b (FN)
	Class>No	c (FP)	d (TN)

$$\text{Accuracy} = \frac{a + d}{a + b + c + d} = \frac{TP + TN}{TP + TN + FP + FN}$$

e.g., The percentage of emails that are being labeled correctly

Issue of Using Accuracy

- Does not deal with imbalance issue well
 - E.g., detecting spam emails where 99% are normal and 1% are spams
 - Given a model that labels every email as normal, the model has 99% accuracy – but is that good? Think about detection of dangerous asteroids
- Lots of classification problems where the classes are skewed (more records from one class than another)
 - Credit card fraud
 - Covid test

Alternative Measures

		PREDICTED CLASS	
ACTUAL CLASS		Class=Yes	Class>No
	Class=Yes	a	b
	Class>No	c	d

Need to specify
what is the
"true" label

$$\text{Precision (p)} = \frac{a}{a + c}$$

e.g., The percentage of emails labeled as "spam" that are actually spam

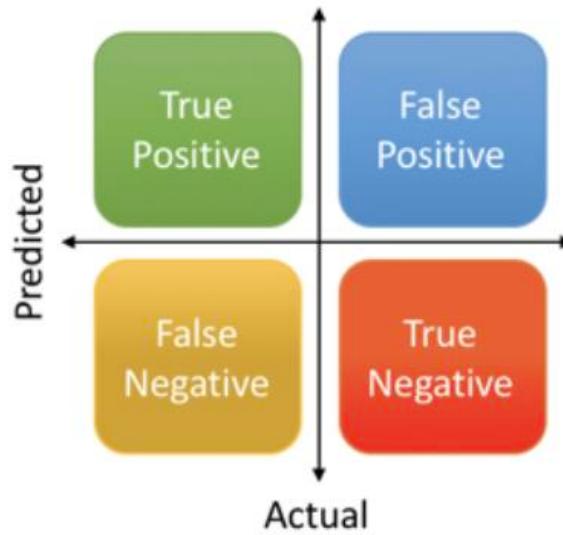
$$\text{Recall (r)} = \frac{a}{a + b}$$

e.g., The percentage of spam emails are labeled as "spam"

$$\text{F - measure (F)} = \frac{2rp}{r + p} = \frac{2a}{2a + b + c}$$

Combines Precision and Recall

Precision vs Recall



$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

- Precision is sensitive to False Positive
- Recall is sensitive to False Negative

Which one is more affordable, False Positive or False Negative?

- Detect if an asteroid is dangerous or not - **High recall**
- Choose stocks to invest – **High precision**

Precision, Recall, F1-score

```
from sklearn import metrics #Import scikit learn metrics module for accuracy calculation

print("Precision is: ", metrics.precision_score(y_test, y_pred, pos_label="Luxury"))

Precision is:  0.8405797101449275

print("Recall is: ", metrics.recall_score(y_test, y_pred, pos_label="Luxury"))

Recall is:  0.8923076923076924

print("Recall is: ", metrics.f1_score(y_test, y_pred, pos_label="Luxury"))

Recall is:  0.8656716417910447
```

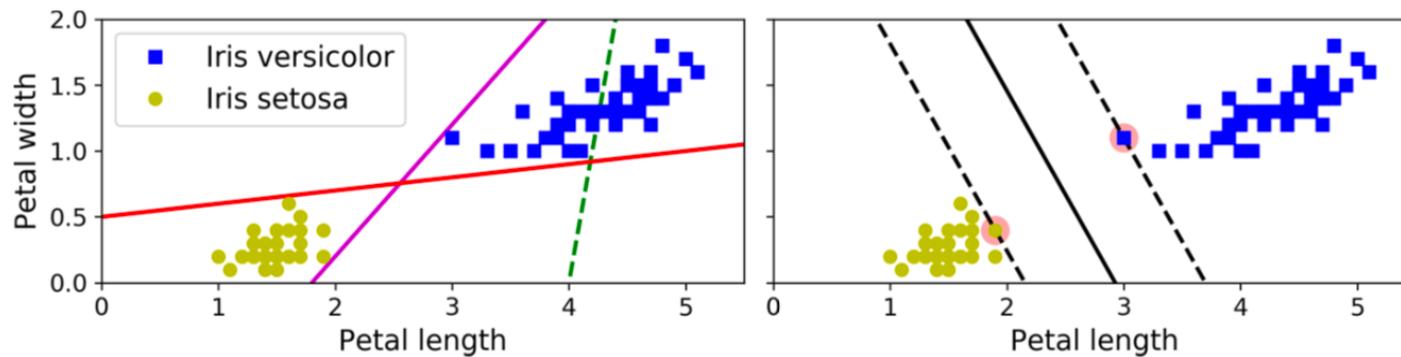
Data-intensive space engineering

Lecture 4

Carlos Sanmiguel Vila

Support Vector Machines

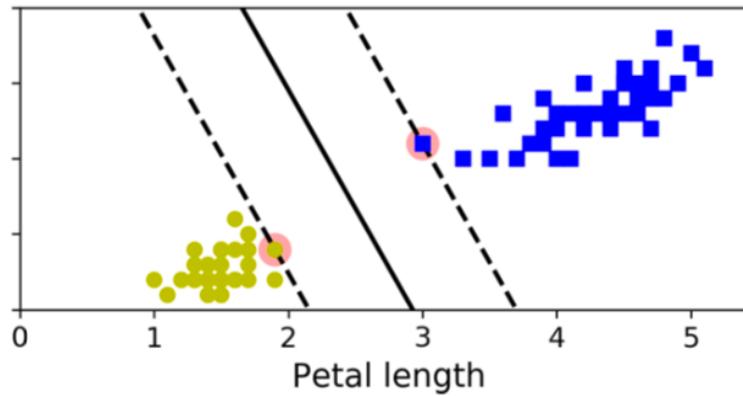
- A support vector machine (SVM) is a machine learning model, capable of performing linear or nonlinear classification, regression, and even novelty detection.
- SVMs shine with small to medium sized nonlinear datasets (i.e., 100/1000 instances), especially for classification tasks. However, they don't scale very well to very large datasets.



- An SVM classifier looks not only to separate the two classes but also to stay as far away from the closest training instances as possible.

Support Vector Machines

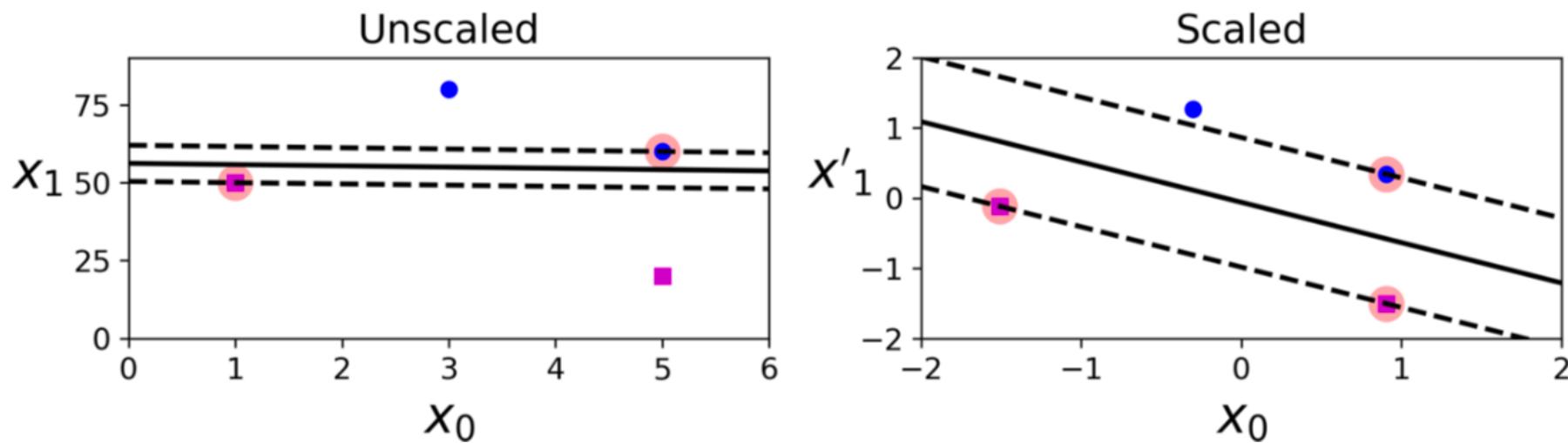
- We look for a large margin classification. An SVM classifier tries to fit the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.



- Notice that adding more training instances “off the street” will not affect the decision boundary at all: it is fully determined (or “supported”) by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure).

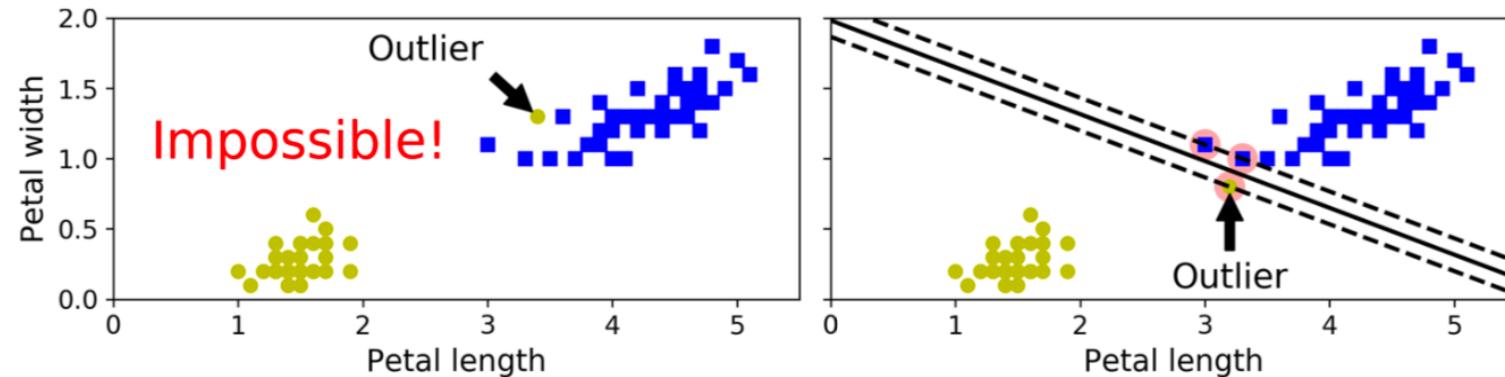
Challenges of using SVMs

- SVMs are very sensitive to the feature scales.



Challenges of using SVMs

- **Hard margin classification** requires that all training instances are correctly classified
- It only works if the dataset is linearly separable
- Sensitive to outliers.



How to solve this problem? -> Soft margin classification

- We use a more flexible model to find a good balance between a large margin and limiting margin violations (i.e., instances that end up in the middle of the street or even on the wrong side).

SVMs in Sklearn

```
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

scaler = StandardScaler()
svm_clf1 = LinearSVC(C=1, loss="hinge", random_state=42)
svm_clf2 = LinearSVC(C=100, loss="hinge", random_state=42)

scaled_svm_clf1 = Pipeline([
    ("scaler", scaler),
    ("linear_svc", svm_clf1),
])
scaled_svm_clf2 = Pipeline([
    ("scaler", scaler),
    ("linear_svc", svm_clf2),
])
scaled_svm_clf1.fit(X, y)
scaled_svm_clf2.fit(X, y)
```

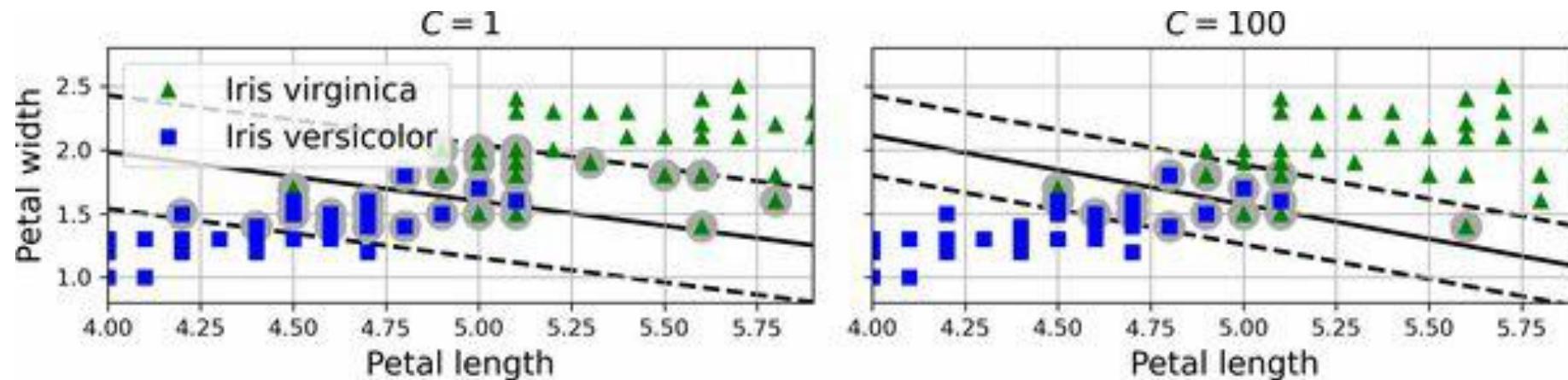
C (Regularization Parameter): Controls the trade-off between maximizing the margin and minimizing classification errors. A small CCC leads to a wider margin with more misclassifications (potentially underfitting), while a large CCC tries to classify all training examples correctly (potentially overfitting).

Hinge Loss is a loss function used primarily in **Support Vector Machines (SVM)** for classification tasks. It's designed to **maximize the margin** between different classes by penalizing misclassified points and those that are too close to the decision boundary.

The hinge loss ensures that the SVM focuses on correctly classifying data points and maximizing the margin between the classes.

SVMs in Sklearn

- The strength of the regularization is inversely proportional to C
- If your SVM model is overfitting, you can try regularizing it by decreasing C
- Reducing C makes the street larger, but it also leads to more margin violations



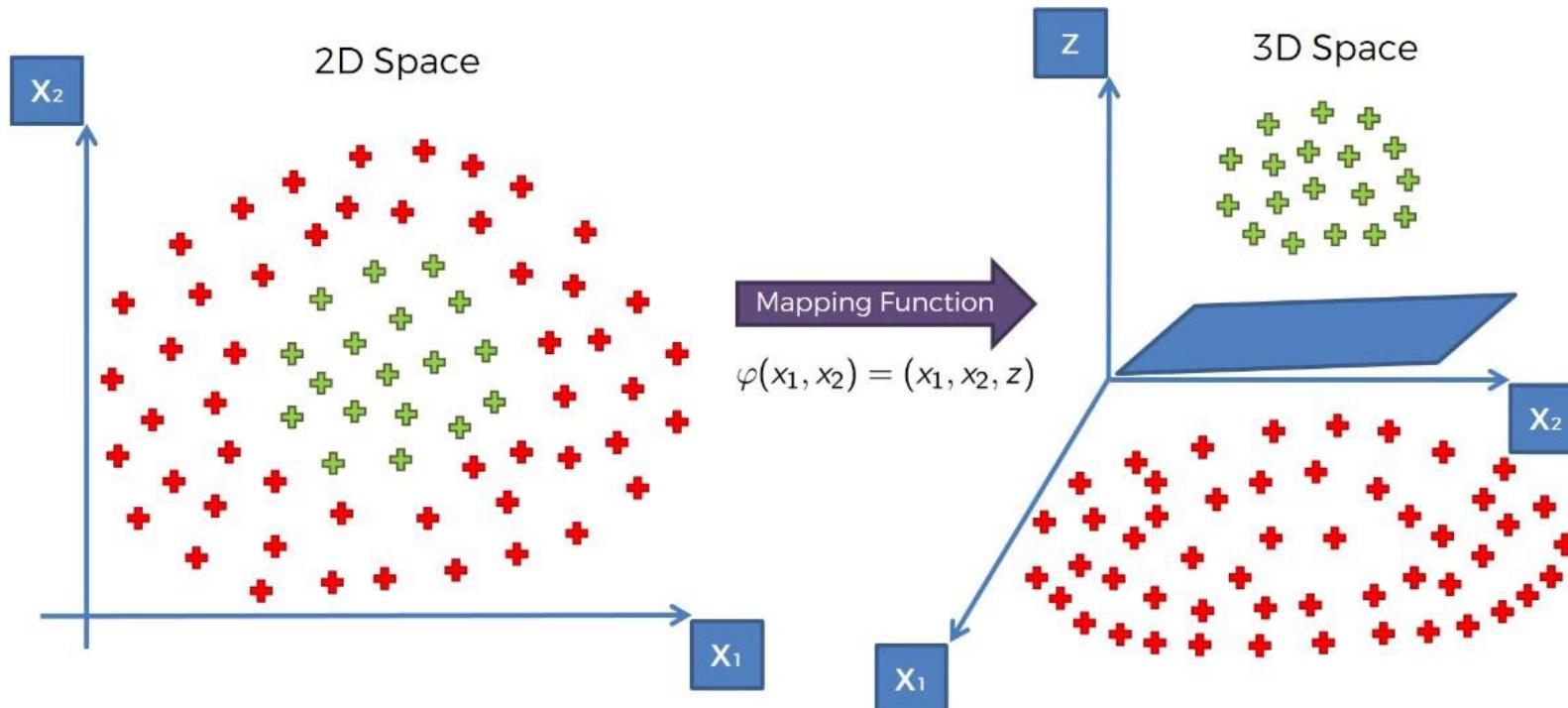
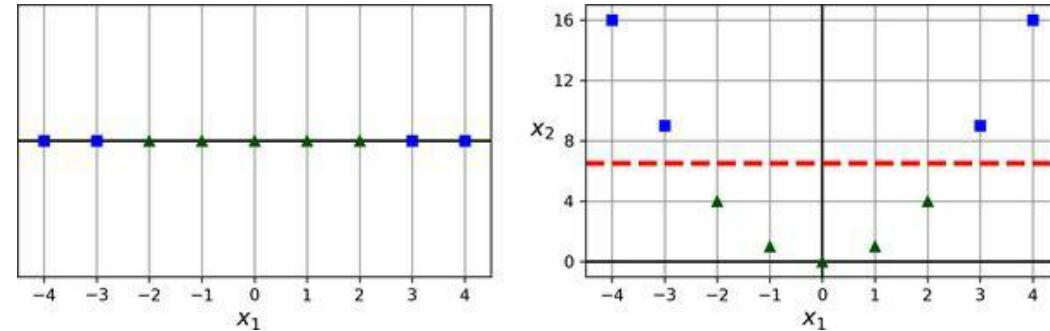
Hinge Loss

$$L(y, f(x)) = \max(0, 1 - y \cdot f(x))$$

Where:

- $y \in \{-1, 1\}$ is the true label of the data point (binary classification).
- $f(x)$ is the predicted value (distance from the decision boundary).
- The loss is zero if $y \cdot f(x) \geq 1$ (correct classification with sufficient margin), and positive otherwise.

Nonlinear SVM Classification



Nonlinear SVM Classification

- In practice, the SVM algorithm is implemented using a kernel, which transforms an input data space into the required form.
- SVM uses a technique called the kernel trick. Here, the kernel transforms a low-dimensional input space into a higher-dimensional space.
- In other words, you can say that it converts non-separable problem to separable problems by adding more dimension to it.
- It is most useful in non-linear separation problems. The kernel trick helps you to build a more accurate classifier.
- A kernel function takes two inputs (data points) and returns the inner product in the new, higher-dimensional space. You don't need to calculate the transformation to the higher-dimensional space explicitly; the kernel function does this indirectly by calculating the distance between two points in that space.

Nonlinear SVM Classification

Here are the common types of kernels:

1. Linear Kernel:

Equivalent to no transformation. This is used when the data is linearly separable.

Inner product: $K(x_i, x_j) = x_i \cdot x_j$

2. Polynomial Kernel:

Maps the original features into a polynomial feature space.

Inner product: $K(x_i, x_j) = (x_i \cdot x_j + 1)^d$, where d is the degree of the polynomial.

3. Radial Basis Function (RBF) Kernel:

Maps the data into an infinite-dimensional space and is widely used for non-linear problems.

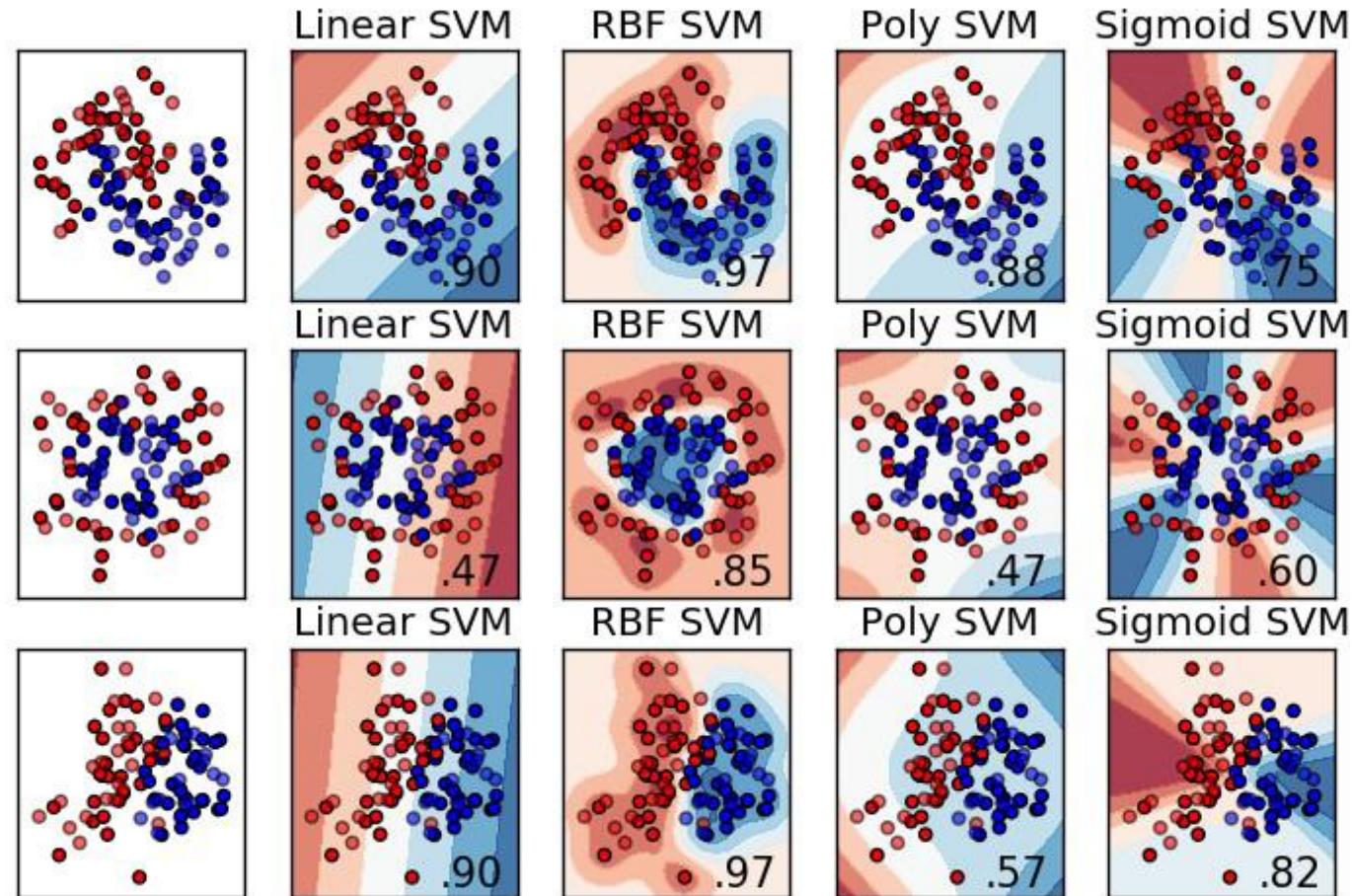
Inner product: $K(x_i, x_j) = e^{-\gamma ||x_i - x_j||^2}$, where γ is a parameter that controls the influence of a training example.

4. Sigmoid Kernel:

Similar to the activation function in neural networks, it can be useful for certain types of binary classification problems.

Inner product: $K(x_i, x_j) = \tanh(\alpha(x_i \cdot x_j) + c)$

Types of kernels

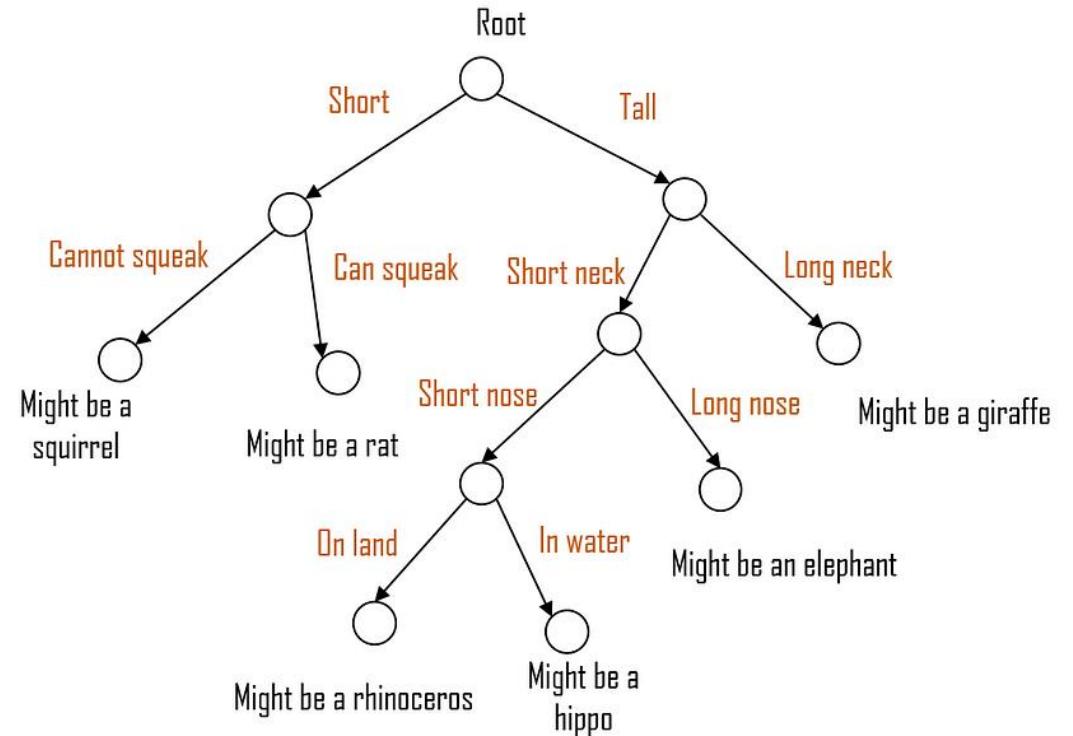


SVM vs. Logistic Regression

Algorithm	Pros	Cons
Logistic Regression	<ul style="list-style-type: none">- Simple and interpretable- Works well for linear separability- Probabilistic outputs	<ul style="list-style-type: none">- Limited to linear decision boundaries- Sensitive to outliers- Struggles with non-linear data
SVM (Linear Kernel)	<ul style="list-style-type: none">- Effective for linearly separable data- Maximizes margin- Robust to outliers	<ul style="list-style-type: none">- Linear decision boundary (same as logistic regression)- Not probabilistic by default
SVM (Non-Linear Kernel)	<ul style="list-style-type: none">- Powerful for non-linearly separable data- Flexible with kernels- Robust to outliers	<ul style="list-style-type: none">- Computationally expensive for large datasets- Requires hyperparameter tuning (e.g., C, gamma)

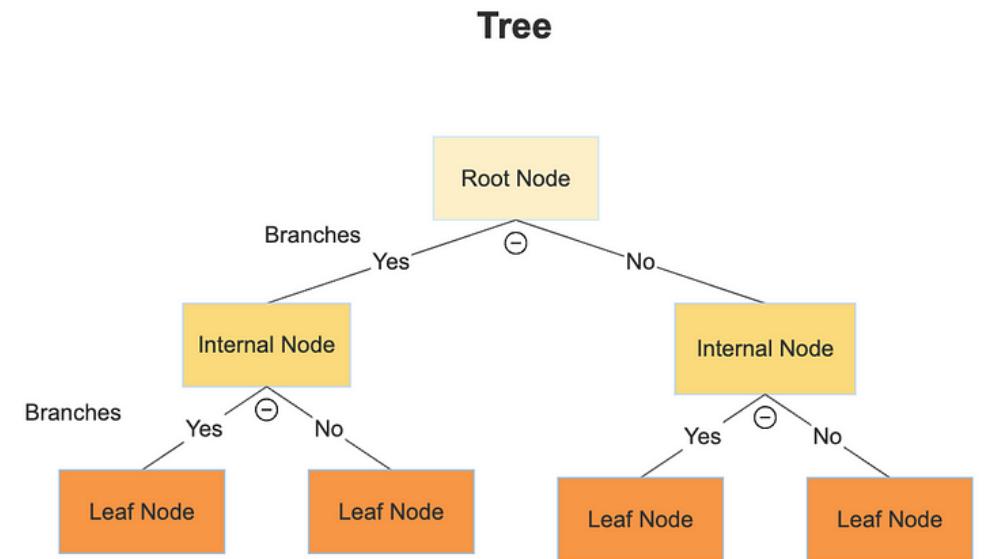
Decision Tree

- A Decision Tree is a supervised machine learning algorithm for classification and regression tasks. It splits the data into subsets based on the feature values, forming a tree-like structure.
- Each internal node represents a "decision" based on a feature, each branch represents the outcome of the decision, and each leaf represents a class or a value for regression.



Structure of a Decision Tree

- **Root Node:** The topmost node representing the entire dataset, which is then divided into two or more homogeneous sets.
- **Decision Nodes:** These are sub-nodes that split further and represent decisions made on the features.
- **Leaf Nodes:** Also known as terminal nodes, these nodes represent the final output or decision, whether it's a class label or a value.
- **Branches:** These are the outcomes of the decision nodes, leading to the next decision node or a leaf node.



How Does a Decision Tree Work?

The process of building a decision tree involves selecting the best attribute to split the data at each node. This selection is based on criteria like Gini impurity, entropy, or variance reduction.

- **Choosing the Best Split:** At each node, the algorithm evaluates all possible splits and selects the one that best separates the data. For classification trees, criteria like Gini impurity or information gain (entropy) are used. For regression trees, the reduction in variance is a common criterion.
- **Splitting:** Once the best split is chosen, the data is divided accordingly, and the process is recursively repeated for each subset.
- **Stopping Criteria:** The tree continues to grow until a stopping criterion is met. This could be a maximum depth, a minimum number of samples in a node, or a threshold for the impurity measure.

Entropy

Entropy measures randomness or uncertainty in a dataset. In the context of decision trees, it quantifies the impurity or disorder in a set of examples.

- If a dataset contains examples from only one class, its entropy is zero, indicating complete purity.
- If the dataset is evenly split between classes, its entropy is at its maximum of 1, indicating maximum disorder.

The formula for entropy (H) for a classification problem is:

$$H(D) = - \sum_{i=1}^C p_i \log_2(p_i)$$

p_i represents the probability of a data point belonging to class i in the dataset D . Specifically:

• p_i is the proportion of data points in the dataset that belong to class i , calculated as:

$$p_i = \frac{|D_i|}{|D|}$$

Where:

- $|D_i|$ is the number of data points in class i .
- $|D|$ is the total number of data points in the dataset.

Gini Impurity

Gini impurity is another measure of impurity used in decision trees. It represents the probability of incorrectly classifying a randomly chosen element if it were labeled according to the distribution of labels in the subset.

Gini impurity is zero when all elements belong to a single class (pure).

It reaches its maximum (0.5 for binary classification) when the elements are equally distributed among the classes.

The formula for Gini impurity (G) is:

$$G(D) = 1 - \sum_{i=1}^C p_i^2$$

Generally, gini impurity should be used in large datasets, and entropy should be used in smaller datasets since entropy has a log function, which makes calculation harder.

Information Gain

Information gain measures the reduction in entropy or impurity after a dataset is split based on an attribute. It quantifies the effectiveness of an attribute in classifying the training data.

- A high information gain indicates that the attribute has effectively split the data into pure subsets.
- It's calculated as the difference between the entropy of the original set and the weighted sum of the entropies of the subsets created by the split.

The formula for information gain (IG) is:

$$IG(D, A) = H(D) - \sum_{v \in V(A)} \frac{|D_v|}{|D|} H(D_v)$$

- $H(D)$: The entropy of the dataset D before the split.
- $H(D_v)$: The entropy of the subset of the dataset D_v after splitting on value v of attribute A .
- $|D_v|$: The number of samples in the subset corresponding to the value v of attribute A .
- $|D|$: The total number of samples in the dataset.

Pre-Pruning and Post-Pruning

Decision trees are powerful tools for classification and regression tasks due to their simplicity and interpretability. However, they are prone to overfitting, especially when they grow too deep and learn the noise in the training data. To combat this, pruning techniques—pre-pruning and post-pruning—are used to improve the model's generalization performance.

What is Pruning?

Pruning in decision trees refers to the process of reducing the size of the tree by removing sections that provide little power in classifying instances. The goal of pruning is to improve the model's performance on unseen data by reducing overfitting.

Pre-Pruning

Pre-pruning, also known as early stopping, involves halting the growth of the decision tree before it becomes too complex. This is done by setting certain conditions that must be met for a split to occur.

Methods of Pre-Pruning

- 1. Maximum Depth:** Limit the maximum depth of the tree. Once the tree reaches this depth, no further splits are made.
- 2. Minimum Samples for Split:** Specify the minimum number of samples required to split a node. If a node has fewer samples than this threshold, it is not split further.
- 3. Minimum Samples per Leaf:** Define the minimum number of samples that a leaf node must have. Nodes that do not meet this criterion are not created.
- 4. Maximum Number of Nodes:** Set a limit on the total number of nodes in the tree. Once this limit is reached, no additional nodes are created.

Pre-Pruning

Advantages of Pre-Pruning

- **Efficiency:** Trees are smaller and less complex, making them faster to construct and evaluate.
- **Reduced Overfitting:** By stopping the growth early, the model is less likely to learn noise from the training data.

Disadvantages of Pre-Pruning

- **Underfitting:** There is a risk of stopping the growth too early, resulting in a model that is too simple and does not capture the underlying patterns in the data.

Post-Pruning

Methods of Post-Pruning

- 1. Reduced Error Pruning:** Remove nodes if the removal improves the performance of the tree on a validation dataset.
- 2. Cost Complexity Pruning (CCP):** Introduce a cost complexity parameter that balances the tree's size and its performance. Nodes are pruned if the cost complexity criterion is improved.
- 3. Minimal Cost-Complexity Pruning:** Calculate the cost complexity for each subtree and prune the tree iteratively to minimize this cost.

Post-Pruning

Advantages of Post-Pruning

- **Better Performance:** By allowing the tree to grow fully before pruning, the model can capture more complex patterns in the data before simplifying.
- **More Control:** Post-pruning allows for more refined adjustments, as the full tree is available for analysis.

Disadvantages of Post-Pruning

- **Computationally Intensive:** Growing a full tree and then pruning it can be more time-consuming and computationally expensive.
- **Complexity:** Requires more sophisticated techniques and validation datasets to determine which nodes to prune.

Decision Trees - Regression

In the context of regression, decision trees aim to predict a continuous target variable. One crucial aspect of decision tree regression is variance reduction, which is used to determine the best splits.

What is Variance?

Remember in statistics, variance measures the spread or dispersion of a set of values. It quantifies how much the values differ from the mean (average) of the dataset. In decision tree regression, variance is used to assess the homogeneity of the target variable within subsets of the data. Lower variance indicates that the values are closer to the mean and, therefore, more homogeneous.

Decision Trees - Regression

Variance Reduction in Decision Trees

Variance reduction is the criterion for selecting the best split at each node in a decision tree regressor. The goal is to minimize the variance within each of the resulting subsets after a split, leading to more accurate predictions.

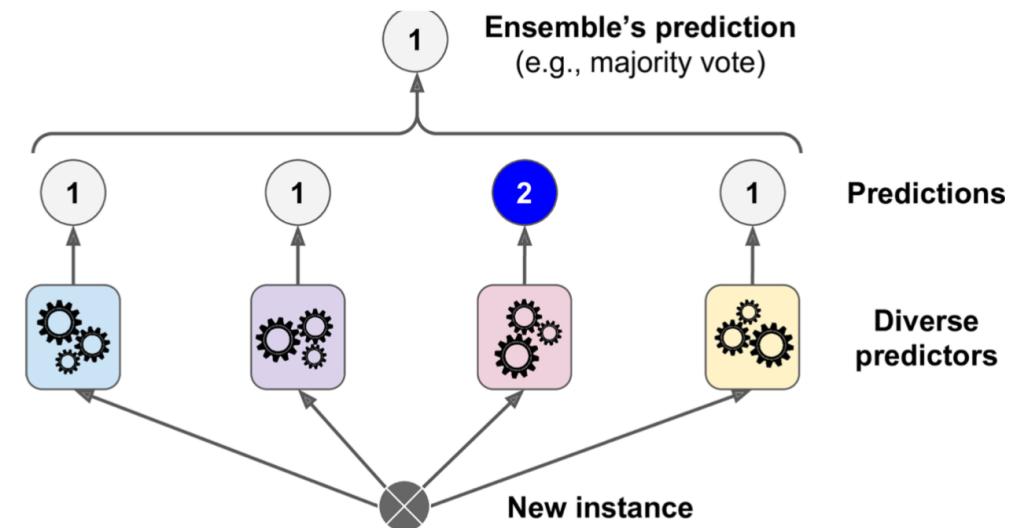
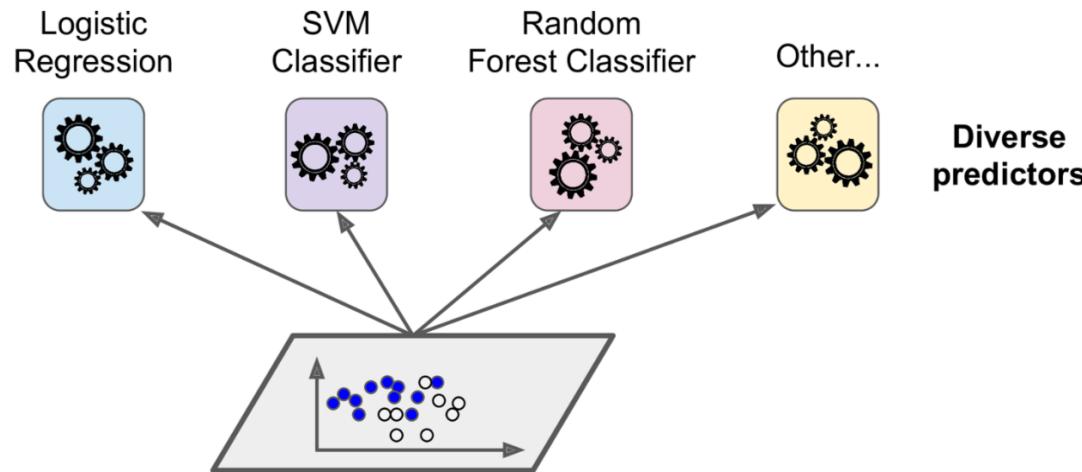
- 1. Before Split:** The variance of the target variable in the parent node is calculated.
- 2. After Split:** The variance of the target variable in each of the child nodes is calculated.
- 3. Variance Reduction:** The reduction in variance is computed as the difference between the variance of the parent node and the weighted sum of the variances of the child nodes. The split that results in the maximum variance reduction is chosen.

Decision Trees

Algorithm	Pros	Cons	When to Use
Decision Trees	<ul style="list-style-type: none">- Easy to interpret and visualize.- No need for feature scaling.- Handles both categorical and continuous data.- Fast for training and prediction.- Can handle non-linear relationships.- Provides feature importance.	<ul style="list-style-type: none">- Prone to overfitting without regularization (max depth, min samples split).- Instability: small changes in data can lead to very different trees.- Generally, lower predictive accuracy than ensemble methods.	<ul style="list-style-type: none">- When interpretability and visualization are important.- Small to medium-sized datasets.- Can be a weak learner in ensemble methods like Random Forests.

Ensemble learning

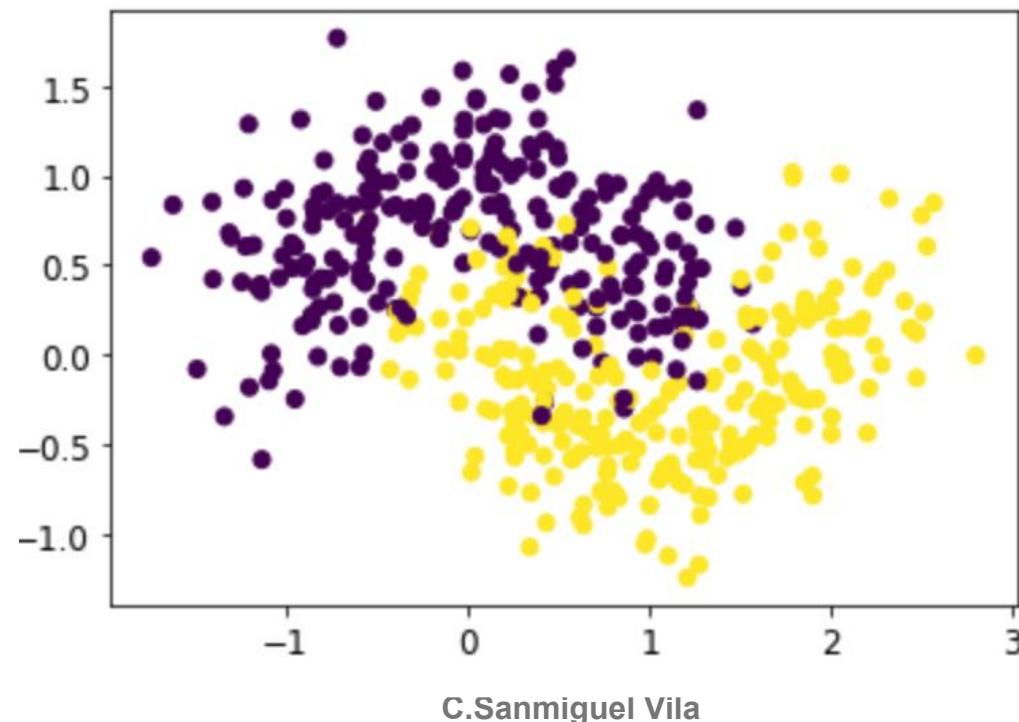
- Ensemble learning is a machine learning technique where multiple models (often called **weak learners**) are trained and combined to solve the same problem.
- The idea is that a group of weak learners can come together to form a **strong learner** that achieves better accuracy and generalization.



Ensemble learning

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```



Ensemble learning

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

log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
svm_clf = SVC(gamma="scale", random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')

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 0.896
VotingClassifier 0.912
```

Why Use Ensemble Learning?

- Increases accuracy by reducing bias and variance.
- Reduces overfitting by averaging multiple models.
- More robust to outliers and noise in the data.

Types of Ensemble Learning:

1. Bagging (Bootstrap Aggregating):

- Builds multiple models in parallel using different subsets of the training data (with replacement).
- Each model votes (for classification) or averages (for regression) to produce the final result.
- Example: **Random Forests**.

2. Boosting:

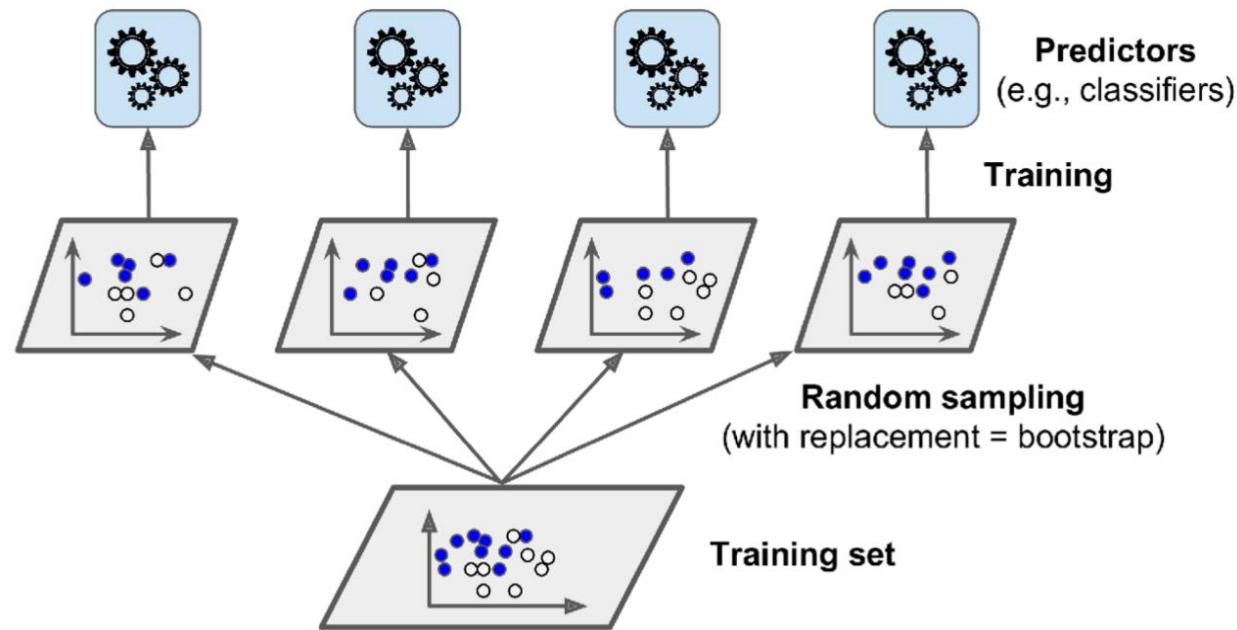
- Models are trained sequentially. Each model tries to correct the errors made by the previous one.
- Example: **AdaBoost, Gradient Boosting**.

3. Stacking:

- Different models are trained, and their outputs are combined using a meta-model that learns how to combine them best.

Bagging and pasting

- In the previous example, we used a diverse set of classifiers
- Another approach is to use the same learning algorithm and train them on different random subsets of the training set
 - Sampling with replacement: bagging
 - Sampling without replacement: pasting



Bagging

- Ensemble of 500 Decision Tree classifiers, each trained on 100 training instances

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

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)

from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))
```

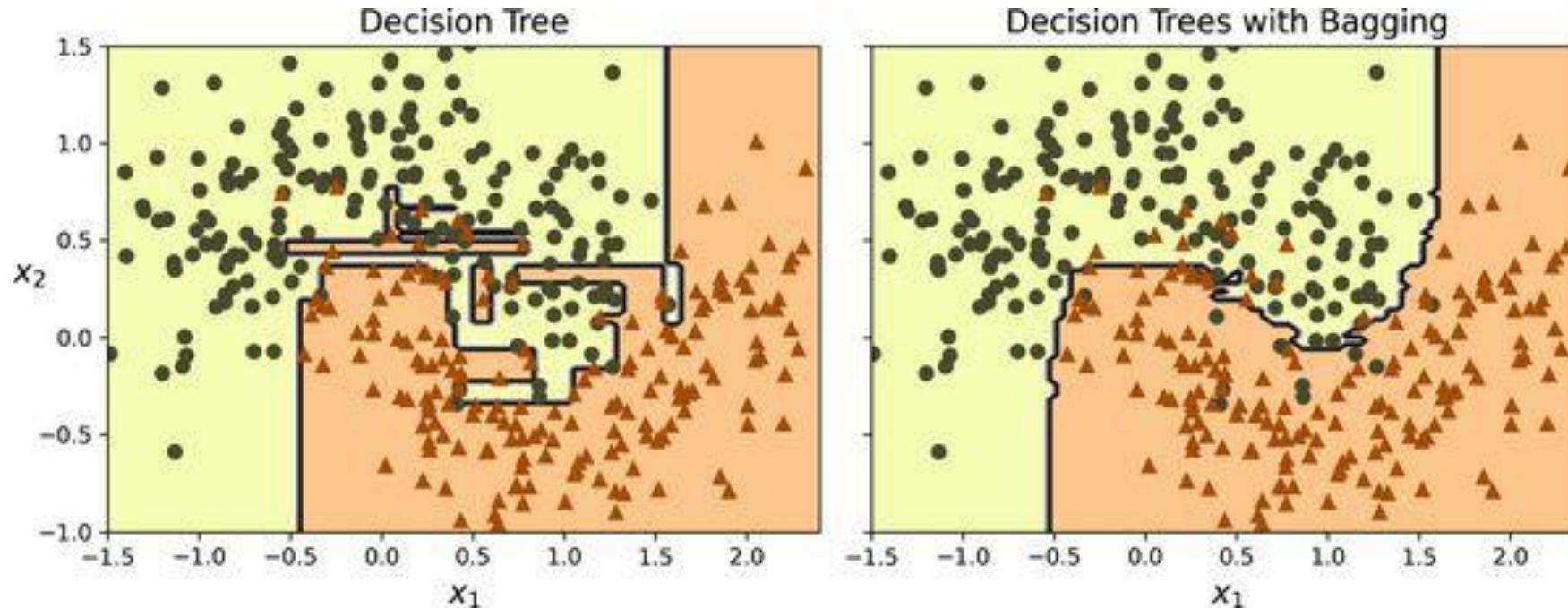
0.904

- Compare with a single Decision Tree classifier

```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))
```

0.856

Bagging



Random Forests

Random Forest is an ensemble method that combines multiple **Decision Trees** to create a stronger model.

It uses **Bagging** and **Feature Randomization** to grow each tree from a random subset of the data and features.

Key Concepts:

- **Bootstrap Sampling:** Each Decision Tree is trained on a random sample of the data with replacement.
- **Random Feature Selection:** At each split in a tree, only a random subset of features is considered. This reduces correlation between trees.
- **Voting:** For classification, the final prediction is the majority vote of all trees. For regression, the output is the average of all trees.

Random Forests

Why Random Forests are Effective

- **Reduces Overfitting:** Because each tree is trained on a different subset of data and features, overfitting is minimized.
- **Feature Importance:** Random Forests provide insights into which features are most important by averaging feature importance across all trees.
- **Handles Missing Data:** Random Forests can handle missing data by averaging over trees that use different data splits.

Random Forests

Hyperparameters in Random Forests

- **n_estimators**: The number of trees in the forest. More trees improve performance, but require more computation.
- **max_depth**: Limits how deep each tree can grow. Limiting depth helps avoid overfitting.
- **max_features**: The number of features to consider when looking for the best split. Lower values reduce overfitting but may limit model capacity.
- **min_samples_split**: The minimum number of samples required to split an internal node.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node.

Random Forests

Advantages	Disadvantages
- Handles large datasets well.	- Can be computationally expensive with many trees.
- Robust to overfitting.	- Interpretability is reduced compared to a single Decision Tree.
- Provides feature importance measures.	- Requires tuning of hyperparameters for best performance.
- Works well with both categorical and continuous data.	- May not perform as well as Boosting in some cases, especially with complex data.

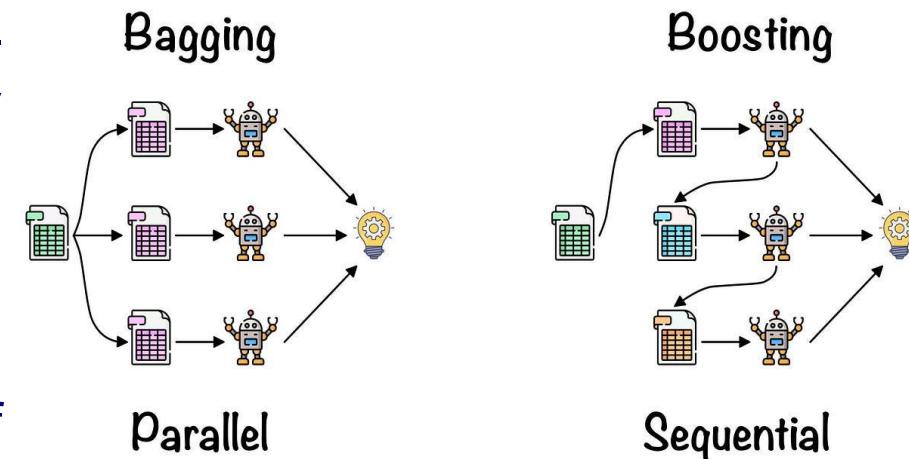
Data-intensive space engineering

Lecture 5

Carlos Sanmiguel Vila

Introduction to Boosting

- **What is Boosting?**
 - Boosting is an ensemble technique where models are trained sequentially, and each model corrects the mistakes of the previous one.
 - The goal of Boosting is to focus on the hardest-to-classify instances, making the model progressively better.
- **How Boosting Differs from Bagging:**
 - Bagging (e.g., Random Forests) builds each model independently in parallel, using different subsets of the data.
 - Boosting builds models sequentially, where each subsequent model focuses on correcting the errors of the previous model.



AdaBoost

- **AdaBoost (Adaptive Boosting):**

- **How it works:** AdaBoost assigns more weight to misclassified instances, ensuring that subsequent models focus on getting those examples right.
- **Key Concept:** Each model in the sequence is trained with adjusted weights based on the errors of the previous model.

AdaBoost Algorithm:

1. Initialize the weights of all samples equally.
2. Train a weak learner (like a shallow Decision Tree).
3. Increase the weights of incorrectly classified samples.
4. Train another weak learner using the updated weights.
5. Repeat until the desired number of models is reached.

AdaBoost (Adaptive Boosting)

- In AdaBoost, the weak learners are usually simple models with low predictive power, such as shallow decision trees or stumps (trees with a single split). Each weak learner is trained sequentially, and at each iteration, the weights of misclassified instances are increased, forcing the model to focus on the difficult-to-classify examples.
- AdaBoost assigns a weight to each weak learner based on their performance, and the final prediction is made by combining the weighted predictions of all weak learners. Instances consistently misclassified by the ensemble receive higher weights, allowing subsequent weak learners to give more emphasis to these challenging cases.
- One of the significant advantages of boosting is its ability to handle complex relationships in the data and significantly improve weak learners' performance. Boosting often outperforms bagging when it comes to reducing both bias and variance. However, boosting is more sensitive to noisy data and outliers than bagging.

Gradient Boosting

What is Gradient Boosting?

- Gradient Boosting is a powerful Boosting method where models are built sequentially, each new model correcting the residual errors made by the previous model.
- Unlike AdaBoost, which adjusts weights, Gradient Boosting optimizes a loss function by adding models that reduce the residuals (the difference between predicted and actual values).

Key Concepts in Gradient Boosting:

1. **Residuals:** The difference between the true value and the predicted value.
2. **Additive Modeling:** New models are added to minimize the residual errors of the previous model.
3. **Gradient Descent:** The models are trained to minimize a specific loss function, and the "gradient" refers to the direction and step size taken to minimize the error.

AdaBoost vs. Gradient Boosting

Aspect	AdaBoost	Gradient Boosting
Boosting Mechanism & Model Focus	Focuses on adjusting weights of misclassified instances at each iteration. Each subsequent model pays more attention to the samples that were misclassified by the previous model.	Focuses on minimizing the residual errors of previous predictions by fitting subsequent models to the residuals. Each subsequent model attempts to correct the residual errors of the previous model using gradient descent.
Loss Function	Implicit loss function (typically based on misclassification error).	Explicit loss function (commonly squared error for regression or log loss for classification).
Weak Learners	Typically shallow decision trees (stumps) (depth = 1).	Typically shallow decision trees , but can use deeper trees.
Error Handling	Errors in classification lead to an increase in the weights of misclassified samples.	Errors are corrected by fitting subsequent models to the residuals of the previous model.
Performance in Noisy Data	Can be sensitive to noisy data since the More robust to noisy data due to its gradient algorithm focuses on misclassified samples, descent-based approach, but can still overfit if potentially overfitting noisy instances.	not regularized properly.

AdaBoost vs. Gradient Boosting

Aspect	AdaBoost	Gradient Boosting
Hyperparameters	Fewer hyperparameters: <code>n_estimators</code> , <code>learning_rate</code> .	More hyperparameters: <code>n_estimators</code> , <code>learning_rate</code> , <code>max_depth</code> , <code>min_samples_split</code> , <code>subsample</code> , etc.
Speed	Generally faster since it deals with fewer parameters and focuses on sample weights.	Generally slower due to the iterative fitting of residuals and more complex hyperparameters to tune.
Regularization	Basic regularization via the number of iterations and learning rate.	More built-in regularization options, including tree depth, learning rate, and subsampling, which can help prevent overfitting.
Parallelization	Harder to parallelize due to the sequential nature of the algorithm (as it relies on or adjusting sample weights).	Can be parallelized at the level of each tree within each boosting iteration (e.g., XGBoost).
Common Use Cases	Often used when fast and simple boosting is required, such as in cases with low noise and requiring robust error correction and better simple weak learners .	Preferred for more complex scenarios and performance with hyperparameter tuning .
Robustness to Outliers	Less robust to outliers because the weight of misclassified points increases.	More robust because outliers can be treated as large residuals, and the model will not overfit them as easily.

XGBoost: An Advanced Gradient Boosting Variant

XGBoost (Extreme Gradient Boosting) is a highly optimized version of Gradient Boosting.

Why is XGBoost Popular?

- **Speed and Efficiency:** XGBoost is optimized for computational speed and memory efficiency. It uses advanced techniques like parallel computing, tree pruning, and cache awareness to train models faster than regular Gradient Boosting. Implements out-of-core computation for large datasets that don't fit into memory.
- **Regularization:** XGBoost includes built-in L1 and L2 regularization, which helps control overfitting and makes the model more robust to noise.
- **Handling Missing Data:** XGBoost can automatically handle missing data by learning which path to take for missing values, making it easier to work with real-world data.
- **Custom Objective Functions:** You can use custom loss functions in XGBoost, making it highly flexible for different types of problems (e.g., ranking, classification, regression).
- **Feature Importance:** XGBoost provides easy access to feature importance, allowing data scientists to interpret which features have the highest impact on model predictions.
- **Scalability:** XGBoost is scalable to large datasets and is often used in machine learning competitions (e.g., Kaggle) and industry.

XGBoost: An Advanced Gradient Boosting Variant

When to Use XGBoost?

- Tabular data with structured rows and columns.
- When you have a large dataset and need a fast and scalable solution.
- When overfitting is a concern, as XGBoost's regularization helps mitigate it.
- When you need a powerful and flexible algorithm for complex problems.

Practical Guide: XGBoost for Beginners

Key Hyperparameters in XGBoost:

- **n_estimators**: The number of trees to be built. Increasing this value can improve model accuracy but also increase training time.
- **learning_rate**: Controls the contribution of each tree. Lower values result in better performance but require more trees.
- **max_depth**: The maximum depth of each tree. Deeper trees can model more complex relationships but are prone to overfitting.
- **subsample**: The fraction of the training data used for building each tree. Lowering this value can prevent overfitting.
- **colsample_bytree**: The fraction of features to consider when building each tree. Reducing this helps prevent overfitting.
- **reg_alpha and reg_lambda**: L1 and L2 regularization terms to control overfitting.

Practical Guide: XGBoost for Beginners

XGBoost Tips for Beginners:

- Start with the default parameters, then gradually tune `n_estimators`, `max_depth`, and `learning_rate`.
- Use cross-validation to avoid overfitting, especially when tuning hyperparameters.
- Regularization is key! Use `reg_alpha` (L1 regularization) and `reg_lambda` (L2 regularization) to control overfitting.
- Use `early_stopping_rounds` to stop training once performance stops improving on a validation set.

CatBoost (Categorical Boosting)

CatBoost (Categorical Boosting) is a gradient boosting algorithm specifically optimized for handling categorical features. Unlike XGBoost, CatBoost can handle categorical data without the need for manual encoding like one-hot encoding or label encoding.

Why is CatBoost Popular?

1. Native Handling of Categorical Features:

1. CatBoost automatically handles categorical features without preprocessing, which is a huge advantage for datasets with high-cardinality categorical variables.

2. Faster Training:

1. CatBoost uses **ordered boosting**, which reduces prediction shift, making the model less prone to overfitting and improving training speed.

3. Robust to Overfitting:

1. Includes several built-in regularization techniques, making it less likely to overfit, especially on small datasets.

4. High Accuracy with Minimal Tuning:

1. CatBoost often requires fewer hyperparameter tuning efforts to achieve competitive accuracy.

CatBoost (Categorical Boosting)

When to Use CatBoost?

- **Datasets with many categorical features**
- When you want to reduce the preprocessing burden and skip manual encoding steps.
- When you are looking for an algorithm that works well out of the box with minimal tuning.
- Particularly useful for **tabular data**, mainly when it contains a mix of categorical and numerical variables.

Practical Guide: CatBoost for Beginners

Key Hyperparameters in CatBoost:

- **iterations**: Number of boosting rounds (similar to n_estimators in XGBoost).
- **depth**: Maximum depth of trees. Shallow trees are faster and less prone to overfitting.
- **learning_rate**: The step size for updates; smaller values often improve accuracy.
- **l2_leaf_reg**: L2 regularization term on the leaf weights to prevent overfitting.
- **subsample**: Fraction of data to use for each tree. Lowering this can reduce overfitting.

Practical Guide: CatBoost for Beginners

CatBoost Tips for Beginners:

- CatBoost often performs well out of the box, but you can start tuning iterations and depth for better accuracy.
- Use verbose=0 to suppress the training output unless you want detailed logs during training.
- CatBoost is particularly effective when working with datasets that have many categorical features.
- Like XGBoost, cross-validation is useful for evaluating model performance and avoiding overfitting.

XGBoost vs CatBoost

- XGBoost is widely used because of its speed, scalability, and flexibility. It performs exceptionally well in large datasets and allows for fine-grained control with its many hyperparameters, making it a go-to for machine learning competitions and production models.
- CatBoost is particularly effective when dealing with categorical features and mixed datasets. It simplifies preprocessing (e.g., no need for manual encoding), and its ordered boosting and automatic handling of categorical variables make it beginner-friendly and robust for business applications without extensive tuning.
- If you're working with tabular data that is mostly numerical, and you need maximum control over model tuning for high performance, start with XGBoost.
- If your dataset contains many categorical variables and you want to avoid the hassle of encoding them, or if you need a model that works well with minimal tuning, CatBoost is a great choice.

Data-intensive space engineering

Lecture 6

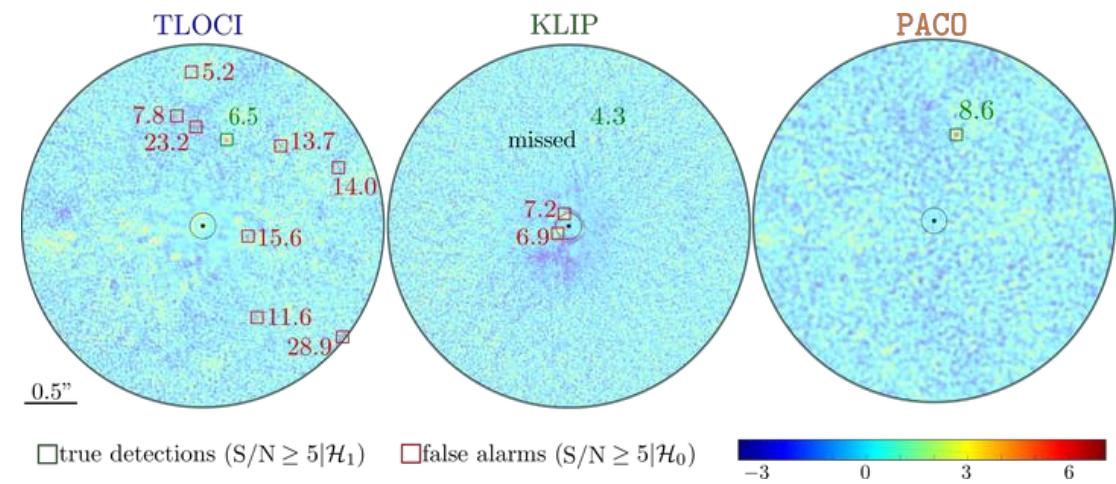
Carlos Sanmiguel Vila

Introduction to Unsupervised

Most available data is unlabeled, i.e., we have the input features X but not the labels y

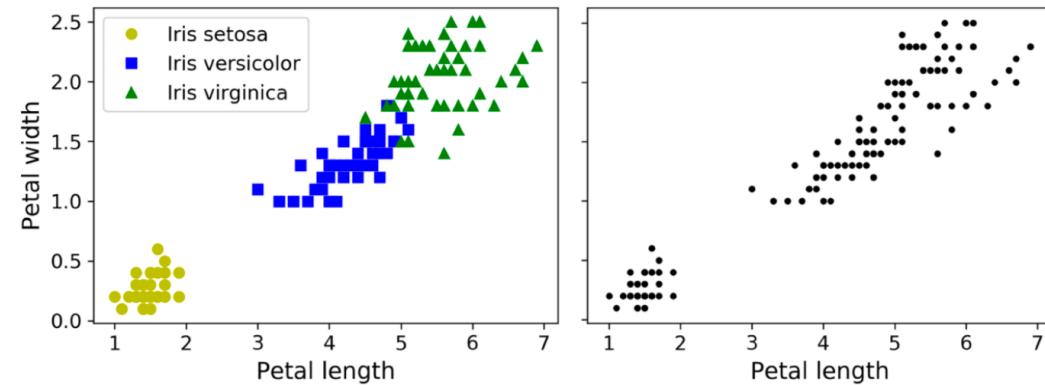
Example of a classical routine:

- Discover new exoplanets by direct imaging
- Take thousands of data every day
- Need to label each observation targeting if an observation is a real candidate for an exoplanet or not to train binary classifiers
- Long, costly, and tedious task



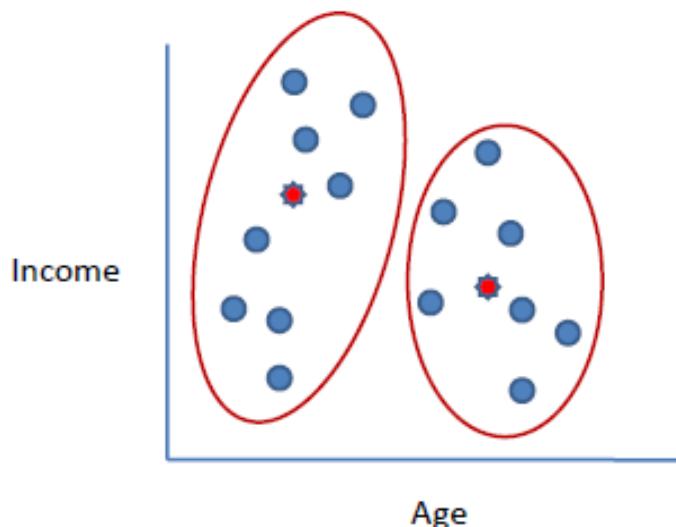
Types of unsupervised learning

- **Clustering**
 - Group similar instances together into clusters
- **Anomaly detection**
 - Learn “what” normal data looks like and then use it to detect abnormal instances
- **Density estimation**
 - Estimating the probability density function of the random process that generated the data set
- **Dimensionality Reduction**
 - Learning to reduce the dimension of our dataset in an efficient manner



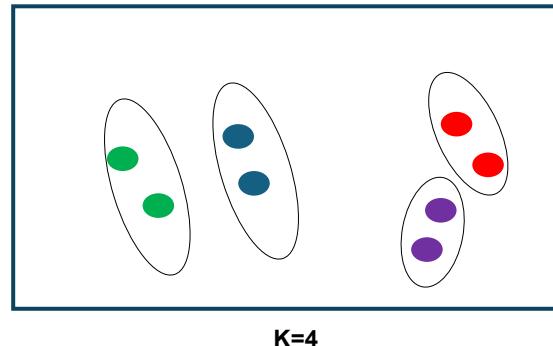
Different clustering algorithms

- There is no universal definition of what a cluster is
 - It depends on the context and different algorithms will capture different kinds of clusters
 - Most typical approach is to look for instances centered around a particular point, called a centroid



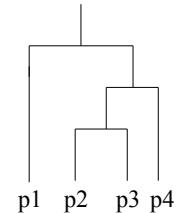
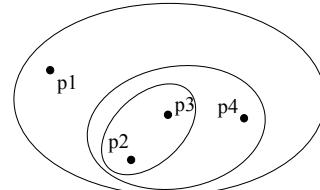
Clustering Types

- Partitional clustering: no overlap between clusters

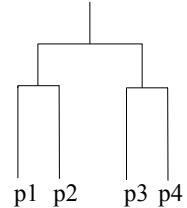
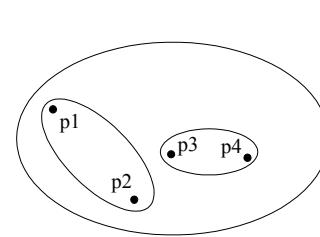


- Hierarchical clustering: clusters are nested

3 clusters: $\{p_1\}, \{p_2, p_3\}, \{p_4\}$



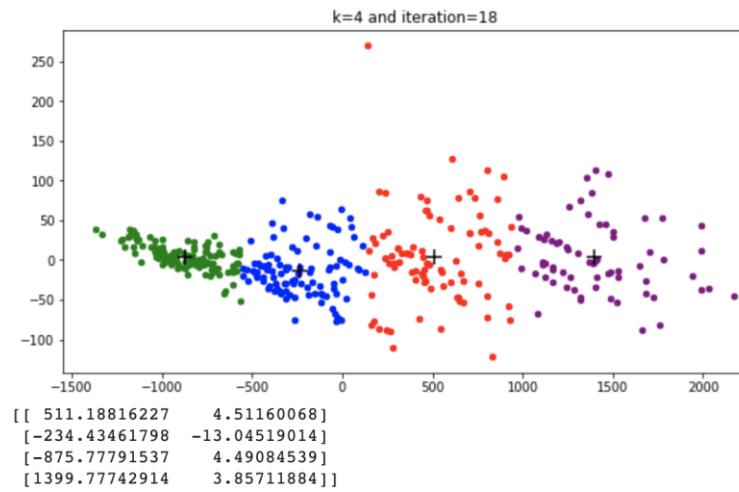
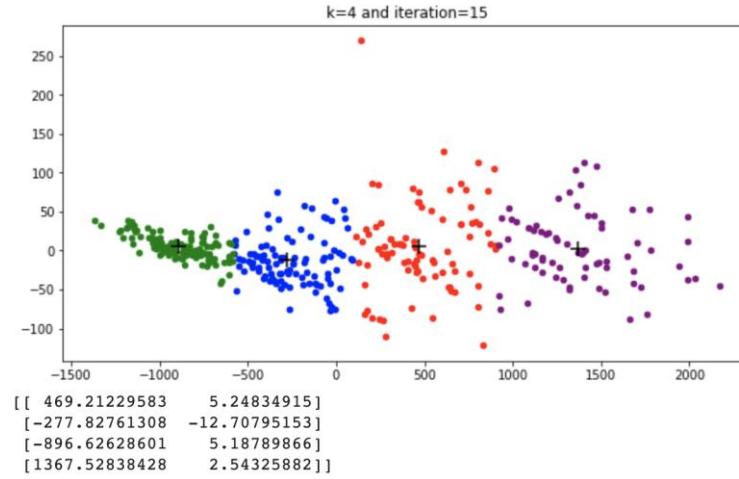
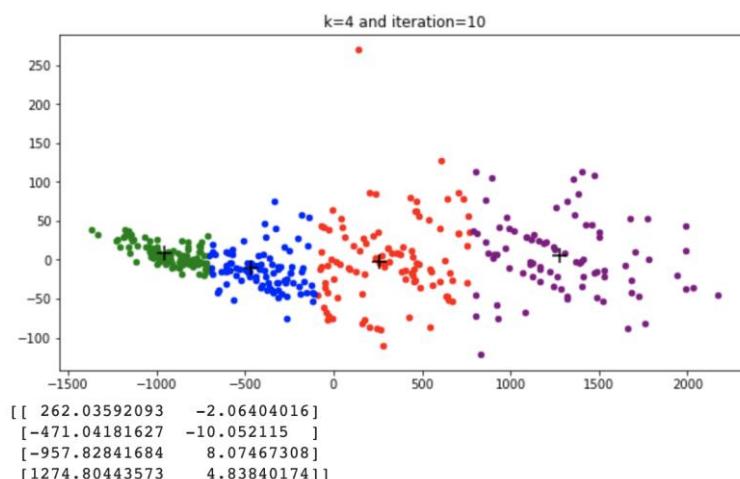
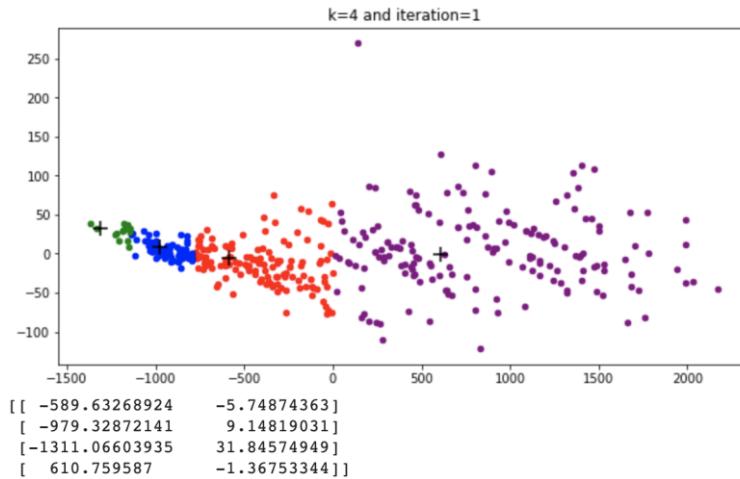
4 clusters: $\{p_1\}, \{p_2\}, \{p_3\}, \{p_4\}$



K-means

- Mostly widely used partitional clustering approach
- Simple and efficient
- Design:
 - Input: a collection of data records, the number of clusters, i.e., K
 - Process:
 1. Randomly choose K points as the initial centroids,
 2. Generate K clusters by assigning all points to the closest centroid
 3. Recompute the centroid of each cluster
 4. Repeat steps 2, 3 until the centroid don't change

K-means



K-means

Advantages:

- Easy to implement and understand.
- Efficient for large datasets with moderate dimensionality.

Disadvantages:

- Sensitive to initial centroid placement (can get stuck in local minima).
- Requires specifying K (number of clusters), which may not always be known beforehand.
- Assumes clusters are spherical and equally sized, which may not hold in real-world applications.

K-means

- **Silhouette Coefficient:**

- This measures how similar a data point is to its own cluster compared to others.

It ranges from **-1 to 1**:

- **+1**: The point is well-matched to its own cluster and far from others.
- **0**: The point is on or very close to the decision boundary between two clusters.
- **-1**: The point might have been misclassified to the wrong cluster.

The **Silhouette Coefficient** helps evaluate the clustering quality and choose the optimal number of clusters (**K**).

K-means

- The **Elbow Method** is a heuristic used to determine the optimal number of clusters K by measuring the inertia (or within-cluster sum of squared errors (SSE)). Inertia represents how tightly the clusters are packed.
- The idea behind the Elbow Method is to run K-Means (or any clustering algorithm) for different values of K and calculate the total SSE (the sum of squared distances between each point and the centroid of its assigned cluster). As K increases, the SSE decreases because the clusters are smaller and better fitted to the data. However, increasing K results in diminishing returns at a certain point, which can be identified by an "elbow" in the plot.

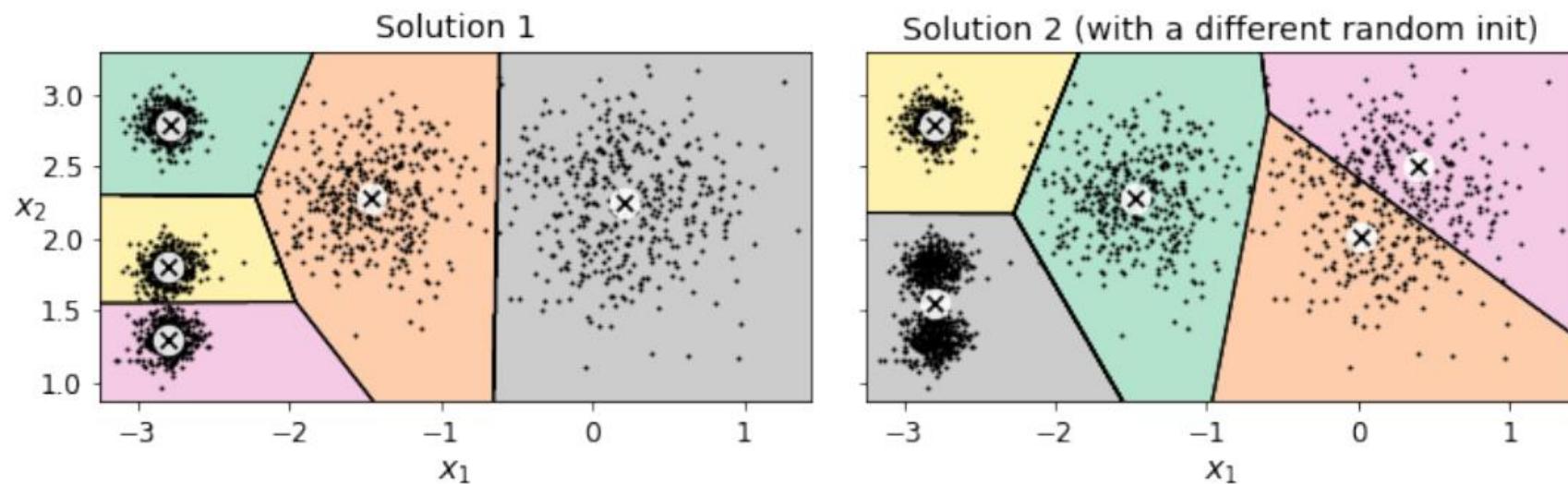
Steps:

- Run K-Means for a range of K values.
- Calculate the SSE for each K. Plot K vs. SSE.
- Identify the "elbow point" where the rate of decrease slows down, suggesting an optimal K.



K-means

- Although the algorithm is guaranteed to converge, it may not converge to the right solution
 - Highly sensitive to the centroid initialization



K-means

- How do we deal with this issue?
 - **k-means++ Initialization:** A smarter initialization method that selects initial centroids that are far apart, improving convergence and cluster quality. By default, scikit-learn's KMeans uses k-means++ initialization.
 - **Multiple Initializations (n_init):** Running K-Means multiple times with different random starts and choosing the best result reduces the risk of poor initialization. The algorithm runs K-Means multiple times with different initializations (based on the number of n_init), and then selects the solution with the lowest within-cluster sum of squares (SSE).
 - **Data Scaling:** Ensures that features with larger values do not dominate distance calculations, improving initialization and clustering results.

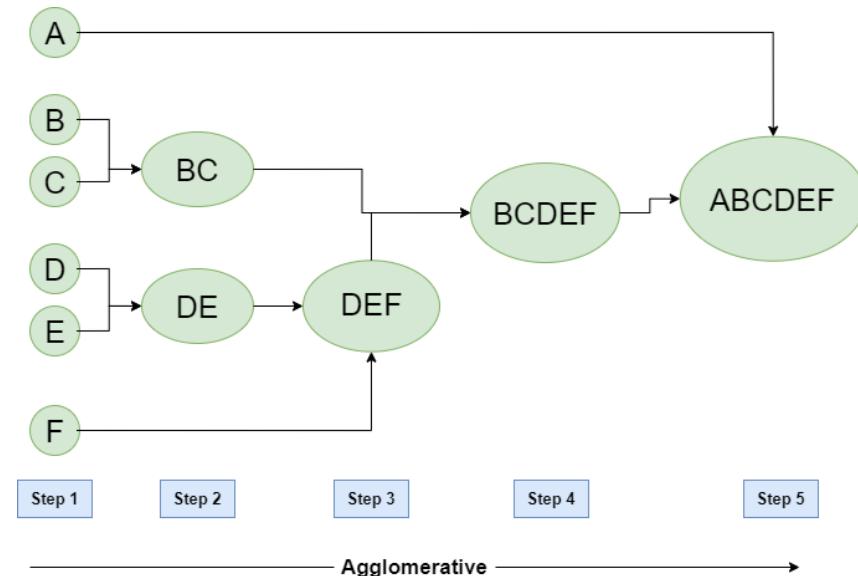
Hierarchical Clustering

- Hierarchical clustering creates a hierarchy of clusters that can be visualized as a **dendrogram**. It is a bottom-up approach (agglomerative), where each data point starts as its own cluster, and pairs of clusters are merged at each step based on their similarity.

- Start with each data point as its own cluster.
- At each iteration, merge the closest clusters.
- Continue merging until only one cluster remains.

- **Linkage Criteria:**

- **Single Linkage:** Distance between the closest points of clusters.
- **Complete Linkage:** Distance between the farthest points of clusters.
- **Average Linkage:** Average distance between points in clusters.
- **Ward's Method:** Minimizes the variance within clusters, often leading to better-defined clusters.



Hierarchical Clustering

Advantages:

- No need to specify the number of clusters.
- Provides a full hierarchy of clusters (can choose any level of granularity).

Disadvantages:

- Computationally expensive for large datasets.
- Sensitive to noise and outliers.

Best Types of Linkage Methods:

- **Single Linkage:** Clusters with irregular shapes, but can result in "chaining" where points get stretched into long chains.
- **Complete Linkage:** Compact, well-separated clusters.
- **Average Linkage:** Balanced, spherical clusters.
- **Ward's Method:** Producing clusters of relatively equal size.

Tip:

- **Ward's Method** is often a good default choice, especially for aerospace applications where clusters are expected to be compact and well-separated (e.g., grouping satellite telemetry phases or system states).

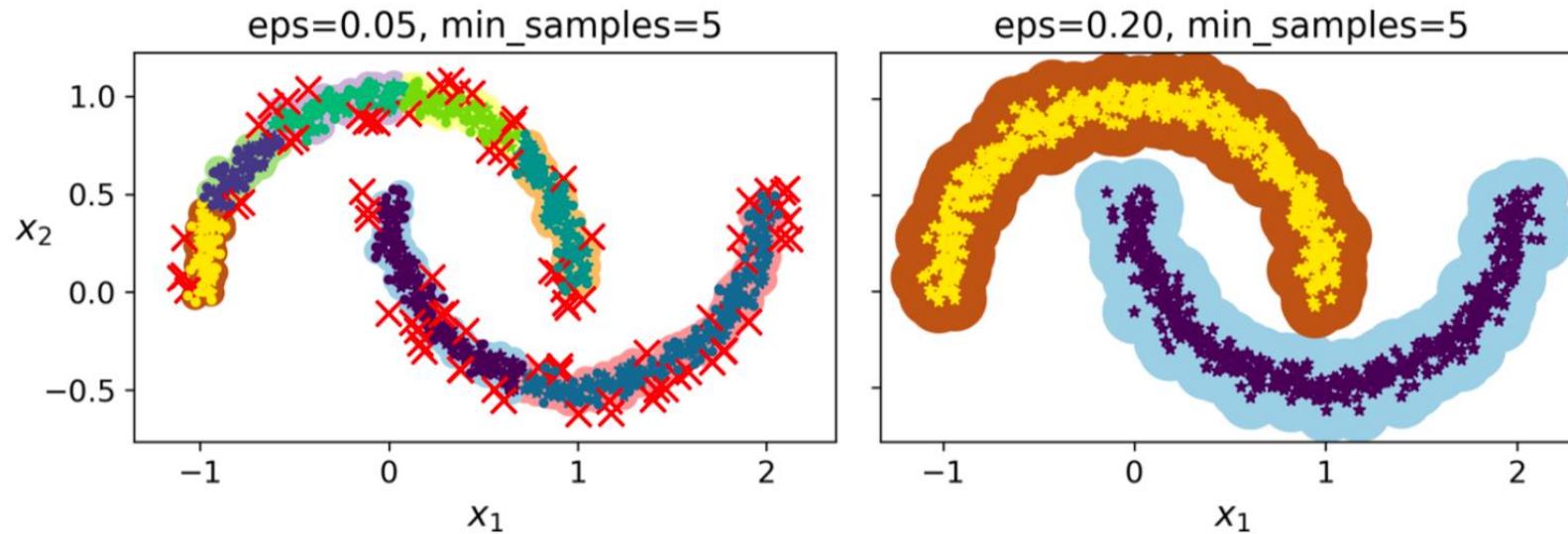
DBSCAN

This algorithm defines clusters as continuous regions of high density

- For each instance, it counts how many samples are located within a small distance ϵ from it (ϵ -neighborhood)
- If an instance has at least `min_samples` samples 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
- Any instance that is not a “core instance” and does not have one in its neighborhood is considered an anomaly

DBSCAN

DBSCAN clustering using two different neighborhood radiuses



- DBSCAN can identify any number of clusters of any shape
- Robust to outliers
- If the density varies significantly, it may be impossible to capture all clusters

What Is Principal Component Analysis?

An unsupervised method that takes X from p dimensions down to k dimensions. PCA is a linear dimensionality reduction technique. It identifies the directions (principal components) in which the data varies the most and projects the data into those directions, reducing the number of features.

Why is this helpful?

- Reduces noise in the data for supervised learning
- Reduces data
- Simpler to visualize data (though dimensions may be unintuitive!)

The Covariance Matrix

Consider some feature $X^{(a)}$, then we can compute the **variance** (*std deviation squared*) of this feature :

$$var(X^{(a)}) = (s^{(a)})^2 = \frac{\sum_{i=1}^n (X_i^{(a)} - \bar{X}^{(a)})^2}{(n - 1)}$$

and we can generalize for $1 \leq a \leq p$ to get a variance matrix

Similarly, we can define **covariance** to capture how the dimensions vary from the mean with respect to each other

$$cov(X^{(a)}, X^{(b)}) = \frac{\sum_{i=1}^n (X_i^{(a)} - \bar{X}^{(a)})(X_i^{(b)} - \bar{X}^{(b)})}{n - 1} = E[(X^{(a)} - \bar{X}^{(a)})(X^{(b)} - \bar{X}^{(b)})]$$

Principal Component Analysis

How it Works:

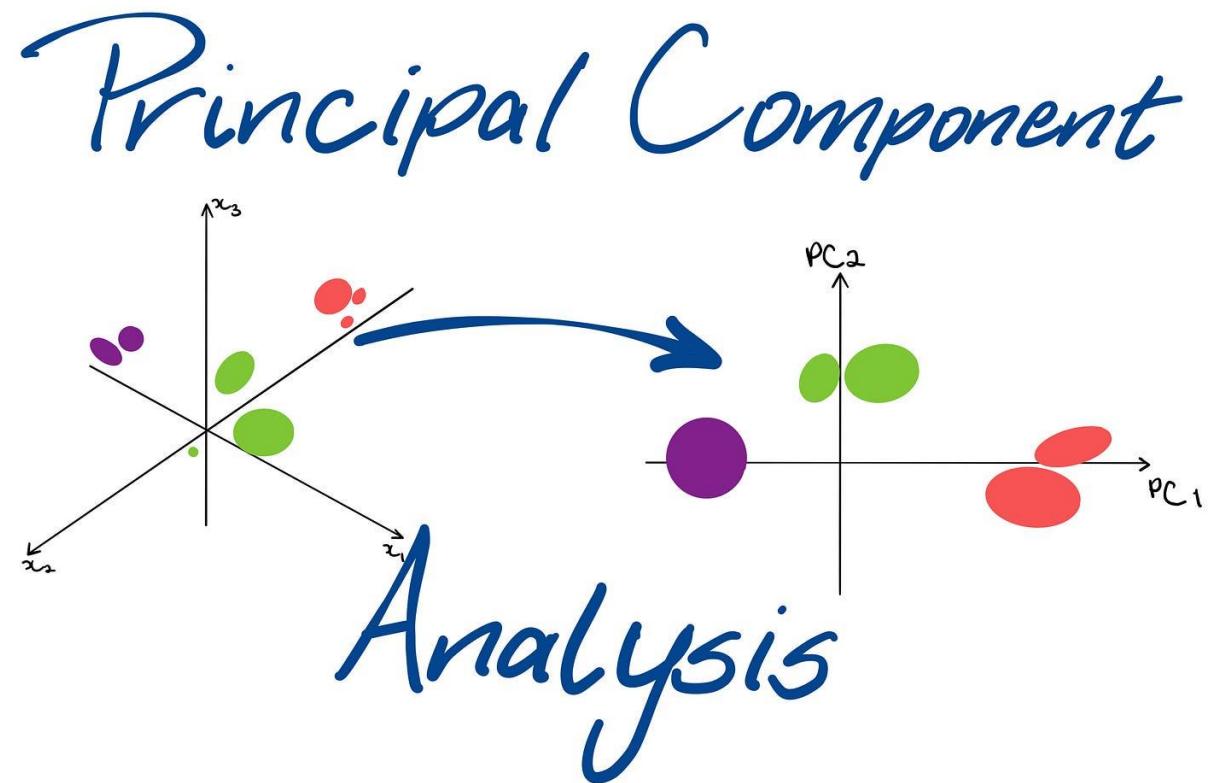
- Compute the covariance matrix of the data.
- Calculate the eigenvectors (principal components) and eigenvalues.
- Sort the eigenvectors by their corresponding eigenvalues in descending order (importance).
- Choose the top k
- Construct a projection matrix W from these k eigenvectors
- Project the original dataset X by multiplying W to obtain a k -dimensional feature subspace Z

$$Z_{n \times k} = X_{n \times p} \cdot W_{p \times k}$$

Principal Component Analysis

Transform to new coordinate system:

- Find directions of maximum variation (covariance)
- Minimize reconstruction error



Principal Component Analysis

Advantages:

- Reduces the dimensionality of data while retaining the most important variance.
- Helps visualize high-dimensional data in 2D or 3D.

Disadvantages:

- PCA is a **linear** method, so it struggles with non-linear datasets.
- The resulting components are often difficult to interpret.

Evaluation:

- **Explained Variance Ratio:** The proportion of the dataset's variance explained by each principal component. A higher variance ratio indicates that more of the data's information is retained.

Isolation Forest

Isolation Forest is an algorithm specifically designed for anomaly detection. Unlike many other algorithms that measure distance or density to detect outliers, it works by isolating observations. It uses the principle that anomalies are few and different, meaning they are easier to isolate than normal points.

How Isolation Forest Works:

- **Isolation:** The basic idea of the Isolation Forest is that anomalies are data points that are "isolated" more quickly than normal points when random cuts are made in the feature space.
- **Random Partitioning:** The algorithm builds an ensemble of **randomly generated decision trees** (called "isolation trees"). At each node in the tree, a feature is randomly selected, and a random split value is chosen between the minimum and maximum value of that feature. This process continues recursively.
- **Path Length:** The number of splits required to isolate a point corresponds to its **path length**. Anomalies are expected to have **shorter path lengths** since they are isolated faster (they are in sparse regions of the data).
- **Ensemble:** The algorithm uses an ensemble of many isolation trees, similar to random forests, to make more robust decisions.

Isolation Forest

Advantages:

- **Effective for Anomaly Detection:** Especially useful for identifying outliers in high-dimensional datasets, making it suitable for aerospace applications like detecting anomalous satellite telemetry data or system malfunctions.
- **No Need for Distance or Density Measures:** Unlike algorithms like DBSCAN or K-Means, Isolation Forest doesn't rely on distance metrics or density estimation, making it effective in scenarios with high-dimensional data.

Disadvantages:

- May require **parameter tuning** (e.g., number of trees, contamination) to optimize performance, although defaults often work well.
- Not as interpretable as some other methods since it's based on random cuts rather than a distance-based approach.

K-Nearest Neighbors (KNN)

- K-Nearest Neighbors (KNN) is a non-parametric and instance-based learning algorithm. It can be used for both classification and regression tasks.
- **For unsupervised learning**, KNN can be adapted for anomaly detection and density estimation. The key idea of KNN is to classify or estimate a data point based on the K nearest neighbours in the feature space.
- **For anomaly detection**, the algorithm identifies points that are far away from their neighbors as outliers.
- **How KNN Works:**
- **Distance-based approach:** The algorithm finds the K nearest neighbors of a given point based on a distance metric, typically Euclidean distance. The distance is used to estimate the density or classify the point.
- **K parameter:** The number of neighbors K plays a crucial role in the performance of the model. A small K is sensitive to noise, while a large K can smooth out important patterns.

K-Nearest Neighbors (KNN)

Advantages

Simple and Intuitive: Easy to understand and implement. **Versatile:** Works for both classification and anomaly detection in high-dimensional data.

No training phase: KNN is an instance-based learner, meaning the algorithm doesn't need explicit training.

Disadvantages

Computationally expensive: KNN requires calculating distances for every point in the dataset during prediction, which can be slow for large datasets.

Sensitive to scaling: Since KNN is distance-based, it's sensitive to feature scaling. Proper normalization or scaling of the data is essential.

Data-intensive space engineering

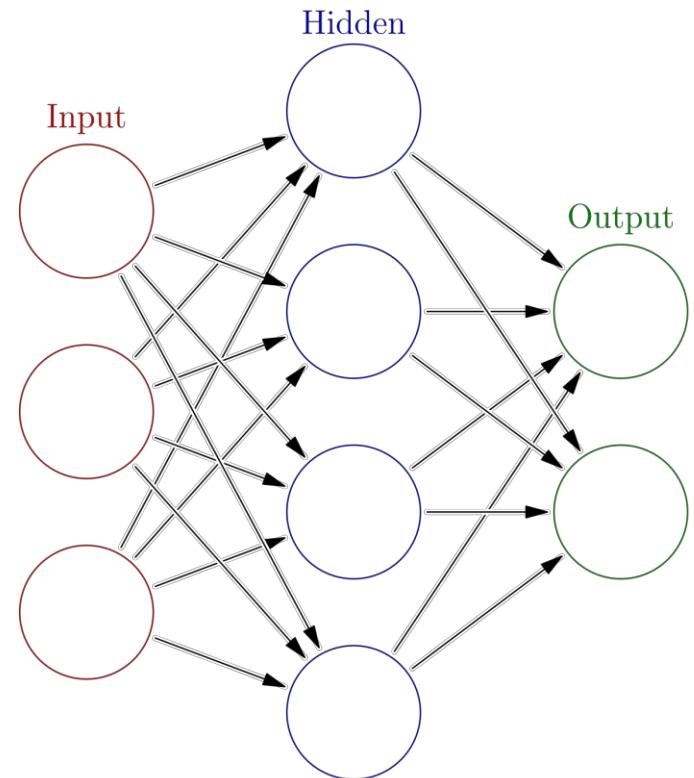
Lecture 7

Carlos Sanmiguel Vila

Introduction to Neural Networks (NNs)

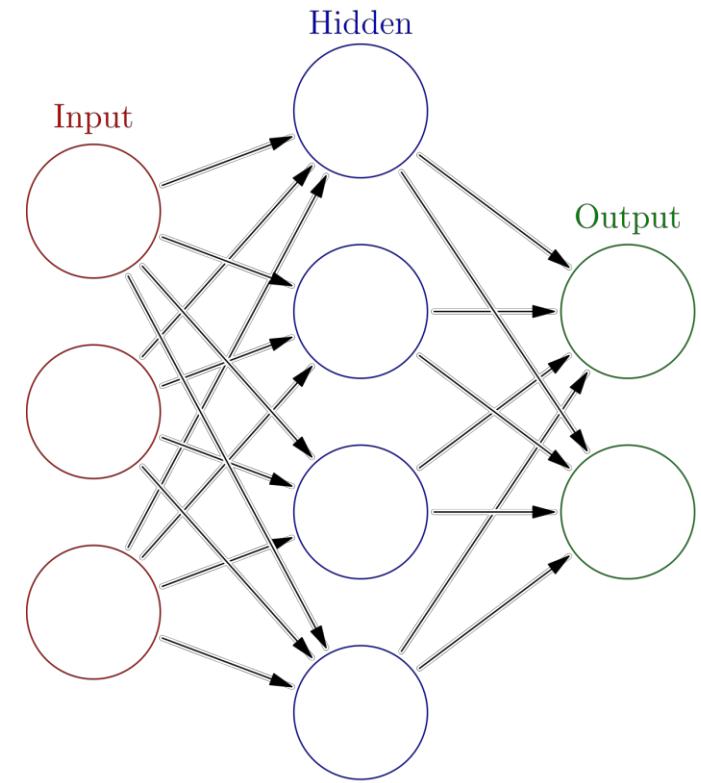
What are Neural Networks?

- A neural network is a computational model inspired by how biological brains work.
- It consists of layers of "neurons" or "nodes" that transform input data through weighted connections.
- The goal is to learn these weights through a process called training, making the model capable of predicting outcomes for unseen data.
- Compare NNs to traditional models (e.g., linear regression)
NNs can capture more complex patterns.



Introduction to Neural Networks (NNs)

- A network is made up of layers: Input Layer, Hidden Layers, and Output Layer.
- Each layer consists of neurons (nodes) that process information through weighted connections.
- **Goal:** To learn the weights of these connections, enabling the network to generalize and make predictions on unseen data.
- Neurons in a network receive inputs, apply weights, and pass the result through an activation function.



Input Layer

The input layer is the first layer of the neural network, where raw data is fed into the model.

Each neuron in this layer corresponds to one feature in the input data.

For example, if you are working with images, each pixel in the image might be a feature, and therefore, each pixel value is passed to the neurons in the input layer.

In a tabular dataset, features could include mass, distance, or coordinates. Each of these features would be represented as a neuron in the input layer.

Example: If your dataset has 10 features (e.g., 10 inputs per data point), your input layer will have 10 neurons.

Key Points: The input layer does not apply any transformation or processing on the data; it simply passes it to the next layer (Hidden Layer).

Hidden Layers

The hidden layers perform transformations on the data using learned weights and biases to extract features and patterns. These are the "internal workings" of the network.

Each neuron in a hidden layer takes the output from the previous layer, applies a weighted sum, adds a bias, and then passes the result through an activation function (like ReLU, Tanh, etc.).

The number of neurons in the hidden layers is flexible. You can have multiple hidden layers (this is known as deep learning when you have more than one hidden layer), and each layer can have different numbers of neurons.

These layers are responsible for learning the complex relationships in the data by adjusting the weights and biases during training.

Hidden Layers

These layers are responsible for learning the complex relationships in the data by adjusting the weights and biases during training.

Neurons in the hidden layers compute $z=W \cdot x + b$ where W are the weights, x is the input, and b is the bias term.

The result z is passed through an activation function like ReLU, Tanh, or Sigmoid, which introduces non-linearity to the model, allowing it to learn more complex patterns.

Example: $a = \text{ReLU}(z)$ (Activation Function)

Multiple hidden layers allow the network to learn more abstract features in the data.

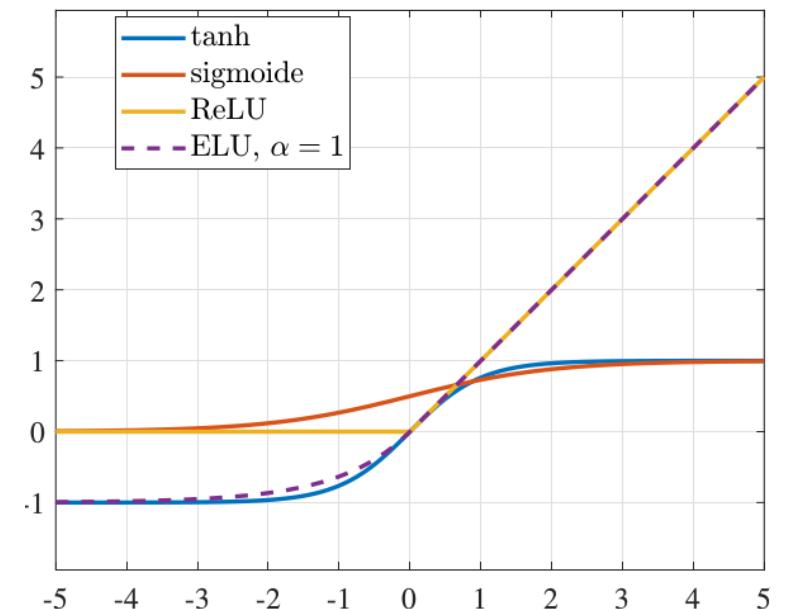
Example: In a neural network trained to classify images of digits (like MNIST), the first hidden layer might learn to detect edges, while deeper hidden layers learn to detect shapes or patterns representing specific digits.

Activation functions

The results obtained from a hidden layer are passed through an activation function to the next layers.

Why Do We Need Activation Functions?

- Activation functions introduce non-linearity to the model, enabling it to capture more complex relationships in data.
- Without an activation function, the neural network would compute a linear transformation, regardless of how many layers it has.



Activation functions

ReLU (Rectified Linear Unit)

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

Advantages: Fast and simple; mitigates the vanishing gradient problem.

Disadvantages: Can cause "dead neurons" where certain neurons stop updating if their output is always 0.

Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Advantages: Smooth and maps values to a range between 0 and 1, making it suitable for binary classification.

Disadvantages: The gradient becomes very small for large positive or negative inputs (vanishing gradient problem).

Activation functions

Tanh (Hyperbolic Tangent)

$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1$$

Advantages: Outputs range between -1 and 1, which helps with the zero-centered output, improving convergence in some cases.

Disadvantages: Also suffers from the vanishing gradient problem for large inputs.

ELU (Exponential Linear Unit)

$$\text{ELU}(x) = x \text{ if } x > 0, \text{ else } \alpha(e^x - 1) \text{ with } 0 < \alpha$$

Advantages: Allows for negative values which help reduce the "dead neuron" problem of ReLU. It can also push mean activations closer to zero, improving learning.

Disadvantages: Computationally more expensive than ReLU.

Output Layer

- The output layer produces the final predictions of the network, either a **classification** or **regression** result.
- The output layer contains neurons that correspond to the target prediction.
- In **regression problems**, the output layer usually has **one neuron** that predicts a continuous value (e.g., predicting a quantity).
- In **classification problems**, the output layer contains **one neuron per class**. For instance, in a binary classification problem, just one output neuron might predict probabilities (using Sigmoid). In multi-class problems, there are usually **multiple neurons**, one for each class, and the network will output probabilities for each class (using Softmax).

Example: In the MNIST classification task, the output layer would have 10 neurons, each representing one of the digits (0–9). The network would predict the probability of the input image belonging to each class (digit).

The **activation function** in the output layer depends on the task:

- **Sigmoid**: For binary classification (e.g., is an email spam or not).
- **Softmax**: For multi-class classification (e.g., classifying digits 0–9).
- **No activation**: For regression tasks, where the output is a continuous value.

Example Flow of a Neural Network

Input Layer: Suppose we have 3 features in our input data (e.g., x_1, x_2, x_3). These values are passed to the neurons in the input layer.

Hidden Layer: Each neuron in the hidden layer receives a weighted sum of the inputs from the input layer, applies an activation function (like ReLU), and passes the result to the next layer.

Output Layer: The final layer takes the processed values from the hidden layer(s) and produces the output:

Classification: If it's a multi-class problem, each neuron in the output layer gives a probability of the input belonging to a particular class.

Regression: If it's a regression problem, the output will be a single continuous value.

Input Layer: Represents the raw data features.

Hidden Layer(s): Learn complex representations through transformations using weights and activation functions.

Output Layer: Provides the final prediction, whether it's a class label or a continuous value.

How Neural Networks Learn – Forward Propagation

In forward propagation, the input data is passed through the network, layer by layer, until it reaches the output.

The network computes a prediction (output) for each input using the following steps:

- Compute the weighted sum for each neuron in the hidden layers: $z=W \cdot x + b$
- Apply the activation function: $a=f(z)$
- Pass the results to the next layer until the output layer is reached.

Objective: To compute a prediction based on the current weights and biases.

How Neural Networks Learn – Loss Function

Why do we need a loss function?

The loss function measures how far the predicted output is from the actual target.

It's a mathematical way to quantify the error between the network's predictions and the true values.

Popular Loss Functions:

Mean Squared Error (MSE): For regression tasks. $L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$

Cross-Entropy Loss: For classification tasks. $L(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^n y_i \log(\hat{y}_i)$

How Neural Networks Learn – Backpropagation and Gradient Descent

What is Backpropagation?

- Backpropagation is the process of computing the gradient of the loss function with respect to each weight in the network, using the chain rule of calculus.
- The gradients tell us how much each weight and bias in the network contributed to the overall error.

How Does It Work?

1. **Forward Pass:** Compute the output using the current weights and compute the loss.
2. **Backward Pass:** Compute the gradient of the loss function with respect to each weight by applying the chain rule.
3. **Gradient Descent:** Use the computed gradients to update the weights and biases in the direction that minimizes the loss.

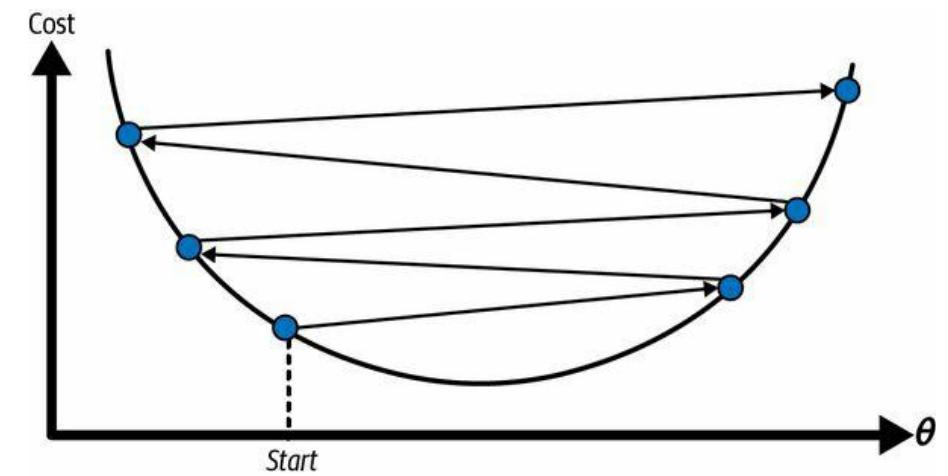
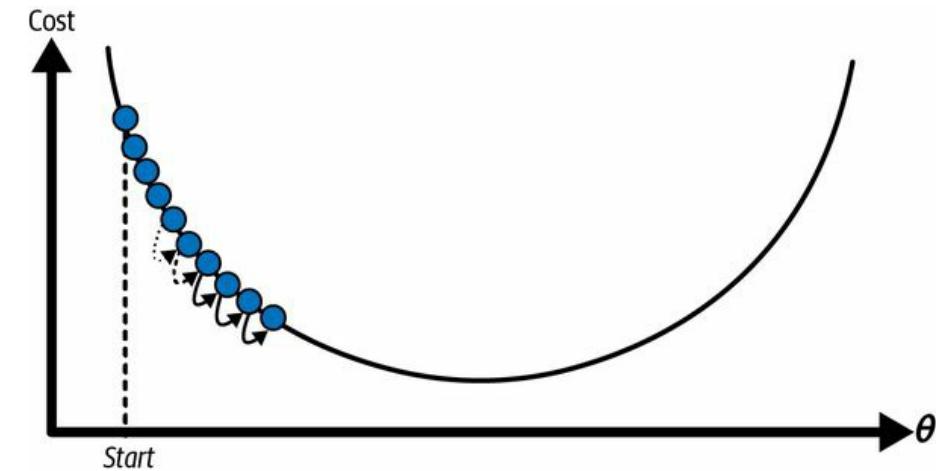
Learning Rate in Gradient Descent

The learning rate η determines the size of the steps we take when updating the weights.

Too Small: The training process will be very slow.

Too Large: The model may overshoot the optimal solution or even fail to converge.

Finding the Right Balance: Experimenting with learning rates is crucial to ensure the model converges efficiently without diverging.



Example: Gradient Update Rule with Two Hidden Layers

Consider a simple neural network with the following structure:

- Input Layer (with 3 features: x_1, x_2, x_3)
- Hidden Layer 1 (with 4 neurons)
- Hidden Layer 2 (with 3 neurons)
- Output Layer (with 1 neuron for a regression task)

The activation function used in both hidden layers is **ReLU**, and there's no activation function in the output layer (because it's a regression task).

Example: Gradient Update Rule with Two Hidden Layers

1. Forward Propagation Equations:

Let's denote:

- W_1 : Weights between **Input Layer** and **Hidden Layer 1**.
- W_2 : Weights between **Hidden Layer 1** and **Hidden Layer 2**.
- W_3 : Weights between **Hidden Layer 2** and **Output Layer**.

Let's compute the forward pass step by step:

Hidden Layer 1:

$$\begin{aligned} z_1 &= W_1 \cdot x + b_1 \\ a_1 &= \text{ReLU}(z_1) \end{aligned}$$

Hidden Layer 2:

$$\begin{aligned} z_2 &= W_2 \cdot a_1 + b_2 \\ a_2 &= \text{ReLU}(z_2) \end{aligned}$$

Output Layer:

$$\begin{aligned} z_3 &= W_3 \cdot a_2 + b_3 \\ y &= z_3 \text{ (no activation function since it's regression)} \end{aligned}$$

Example: Gradient Update Rule with Two Hidden Layers

2. Loss Function:

Assume we're using **Mean Squared Error (MSE)** as the loss function for this regression task. For a single sample:

$$L(y, \hat{y}) = \frac{1}{2} (y - \hat{y})^2$$

Where y is the true value, and \hat{y} is the predicted value from the network.

3. Backpropagation and Gradient Update:

During backpropagation, we compute the **gradients** of the loss function with respect to each weight matrix W_1, W_2, W_3 . This is done using the **chain rule** of calculus.

The **gradients** are used to update the weights in the direction that minimizes the loss:

$$W_i \leftarrow W_i - \eta \frac{\delta L}{\delta W_i}$$

Where η is the learning rate, and $\frac{\delta L}{\delta W_i}$ represents the gradient of the loss with respect to the weight matrix W_i

Example: Gradient Update Rule with Two Hidden Layers

During backpropagation, we compute the gradients of the loss with respect to each weight and bias.

Output Layer (backpropagation step for W_3):

$$\frac{\delta L}{\delta z_3} = y - \hat{y} \quad \frac{\delta L}{\delta W_3} = \frac{\delta L}{\delta z_3} a_2 \quad \frac{\delta L}{\delta b_3} = \frac{\delta L}{\delta z_3}$$

Hidden Layer 2 (backpropagation step for W_2):

$$\frac{\delta L}{\delta z_2} = \frac{\delta L}{\delta z_3} W_3 \text{ReLU}'(z_2) \quad \frac{\delta L}{\delta W_2} = \frac{\delta L}{\delta z_2} a_1 \quad \frac{\delta L}{\delta b_2} = \frac{\delta L}{\delta z_2}$$

Here, $\text{ReLU}'(z_2)$ is the derivative of the ReLU activation function. Since ReLU is 0 for negative inputs and 1 for positive inputs, the gradient either passes through or is set to 0.

Example: Gradient Update Rule with Two Hidden Layers

Backpropagation and Gradient Calculation

Hidden Layer 1 (backpropagation step for W_1):

$$\frac{\delta L}{\delta z_1} = \frac{\delta L}{\delta z_2} W_2 \text{ReLU}'(z_1) \quad \frac{\delta L}{\delta W_1} = \frac{\delta L}{\delta z_1} x \quad \frac{\delta L}{\delta b_1} = \frac{\delta L}{\delta z_1}$$

Gradient Descent (Updating Weights and Biases)

Now, we use the computed gradients to update the weights and biases in each layer using **gradient descent**. The update rule is:

$$W_i \leftarrow W_i - \eta \frac{\delta L}{\delta W_i}$$

The same process applies for biases:

$$b_i \leftarrow b_i - \eta \frac{\delta L}{\delta b_i}$$

Example: Gradient Update Rule with Two Hidden Layers

Summary of the Process:

- **Forward Propagation:** Input flows through each layer, applying weights, biases, and activation functions to compute the prediction.
- **Loss Computation:** The loss function calculates how far off the prediction is from the actual target.
- **Backpropagation:** The gradients of the loss with respect to each weight and bias are computed layer by layer, starting from the output and working backwards.
- **Gradient Descent:** Weights and biases are updated based on the gradients using the gradient descent update rule.

How Neural Networks Learn – Backpropagation and Gradient Descent

What is Backpropagation?

- Backpropagation is the process of computing the gradient of the loss function with respect to each weight in the network, using the chain rule of calculus.
- The gradients tell us how much each weight and bias in the network contributed to the overall error.

How Does It Work?

1. **Forward Pass:** Compute the output using the current weights and compute the loss.
2. **Backward Pass:** Compute the gradient of the loss function with respect to each weight by applying the chain rule.
3. **Gradient Descent:** Use the computed gradients to update the weights and biases in the direction that minimizes the loss.

Why Use PyTorch for Neural Networks?

- PyTorch is one of the most popular deep learning libraries.
- It allows for: Dynamic Computation Graphs (easy debugging and flexibility).
- GPU Support for faster computations.
- Pythonic syntax (works well with other Python libraries like NumPy).
- Real-World Adoption: Used by companies like Facebook, Tesla, and Microsoft.



Setting Up PyTorch

In Google Colab (with GPU access):

```
!pip install torch torchvision
```

Test the installation:

```
import torch
x = torch.rand(5, 3)
print(x)
```

Verify that PyTorch detects the GPU:

```
if torch.cuda.is_available():
    print("GPU is available")
else:
    print("Using CPU")
```

Introduction to Tensors

Tensors are the basic building blocks in PyTorch (similar to NumPy arrays but with GPU support).
Example:

```
tensor = torch.tensor([[1, 2], [3, 4]])
print(tensor)
```

Basic Operations:

```
y = tensor + tensor
print(y)
```

Comparing Activation Functions in PyTorch

```
import torch
import torch.nn.functional as F

# Example input tensor
x = torch.tensor([-2.0, -1.0, 0.0, 1.0, 2.0])

# Applying different activation functions
relu_output = F.relu(x)
sigmoid_output = torch.sigmoid(x)
tanh_output = torch.tanh(x)
elu_output = F.elu(x)

print("ReLU Output:", relu_output)
print("Sigmoid Output:", sigmoid_output)
print("Tanh Output:", tanh_output)
print("ELU Output:", elu_output)
```

Example Neural Network in PyTorch

Steps to Define a Neural Network in PyTorch:

1. Define a model class by inheriting from nn.Module.
2. Create layers inside the `__init__()` method.

```
import torch
import torch.nn as nn

# Defining the network class
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 50) # Input to hidden layer (10 input features, 50 n
        self.fc2 = nn.Linear(50, 1) # Hidden to output layer (1 output for regressio

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # Apply ReLU activation function on the first lay
        x = self.fc2(x)           # Linear output for regression
        return x
```

Example Neural Network in PyTorch

Steps to Define a Neural Network in PyTorch:

3. Define the forward pass in the forward() method.
4. Use optimizers and loss functions for training.

```
model = SimpleNN() # Create an instance of the network
inputs = torch.randn(64, 10) # Random batch of 64 samples with 10 features each
outputs = model(inputs) # Forward pass through the network
print(outputs.shape) # Should output (64, 1) since we have 1 output neuron
```

```
criterion = nn.MSELoss() # Loss function for regression
targets = torch.randn(64, 1) # Random target values for the batch
loss = criterion(outputs, targets) # Compute the loss
print(loss)
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
optimizer.zero_grad() # Clear the previous gradients
loss.backward() # Backpropagation: compute gradients of the loss
optimizer.step() # Update weights based on gradients
```

Example Neural Network in PyTorch

Now let's integrate all the pieces into a training loop

Key concepts:

- **Epoch:** One full pass through the entire training dataset.
- **Batch:** A subset of the dataset. Instead of passing the entire dataset at once, we break it into batches.
- **Iteration:** Each time a batch is processed, it counts as one iteration.

```
import matplotlib.pyplot as plt

# List to store loss values for each epoch
loss_values = []

for epoch in range(100): # Train for 100 epochs
    inputs = torch.randn(64, 10) # Batch of inputs
    targets = torch.randn(64, 1) # Batch of target values

    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)

    # Backward pass and optimization
    optimizer.zero_grad() # clear gradients
    loss.backward() # Backpropagation
    optimizer.step() # Update weights

    # Store the loss value for visualization
    loss_values.append(loss.item())

    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Loss: {loss.item()}')

# Plotting the loss over epochs
plt.plot(range(100), loss_values)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Over Time')
plt.show()
```

Example Neural Network in PyTorch

Steps to Define a Neural Network in PyTorch:

3. Define the forward pass in the forward() method.
4. Use optimizers and loss functions for training.

```
model = SimpleNN() # Create an instance of the network
inputs = torch.randn(64, 10) # Random batch of 64 samples with 10 features each
outputs = model(inputs) # Forward pass through the network
print(outputs.shape) # Should output (64, 1) since we have 1 output neuron
```

```
criterion = nn.MSELoss() # Loss function for regression
targets = torch.randn(64, 1) # Random target values for the batch
loss = criterion(outputs, targets) # Compute the loss
print(loss)
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
optimizer.zero_grad() # Clear the previous gradients
loss.backward() # Backpropagation: compute gradients of the loss
optimizer.step() # Update weights based on gradients
```

Example Neural Network in PyTorch

```
import torch.nn as nn

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 50) # Input layer to hidden layer
        self.fc2 = nn.Linear(50, 1) # Hidden layer to output

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # ReLU activation
        x = self.fc2(x)           # Linear output for regression
        return x

model = SimpleNN()
```

Example Neural Network Training

```
import torch
import torch.nn as nn
import torch.optim as optim

# Simple Neural Network (with 1 hidden layer)
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 50) # Input to hidden layer
        self.fc2 = nn.Linear(50, 1) # Hidden to output layer

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # ReLU activation
        x = self.fc2(x)           # No activation in the output (regression)
        return x

# Create the model, define the loss function and the optimizer
model = SimpleNN()
criterion = nn.MSELoss() # Using Mean Squared Error Loss for regression
optimizer = optim.SGD(model.parameters(), lr=0.01) # Using stochastic gradient descent
```

 Copiar código

Example Neural Network Training

```
# Example training loop (simplified)
for epoch in range(100):
    inputs = torch.randn(64, 10) # Random batch of 64 samples
    targets = torch.randn(64, 1) # Random target values

    # Forward pass
    predictions = model(inputs)
    loss = criterion(predictions, targets)

    # Backward pass
    optimizer.zero_grad()      # Zero out the previous gradients
    loss.backward()             # Compute gradients
    optimizer.step()           # Update the weights

    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item()}")
```

Data-intensive space engineering

Lecture 8

Carlos Sanmiguel Vila

Improving NNs Training

In this lecture, we review a few techniques to improve the performance of (deep) neural networks

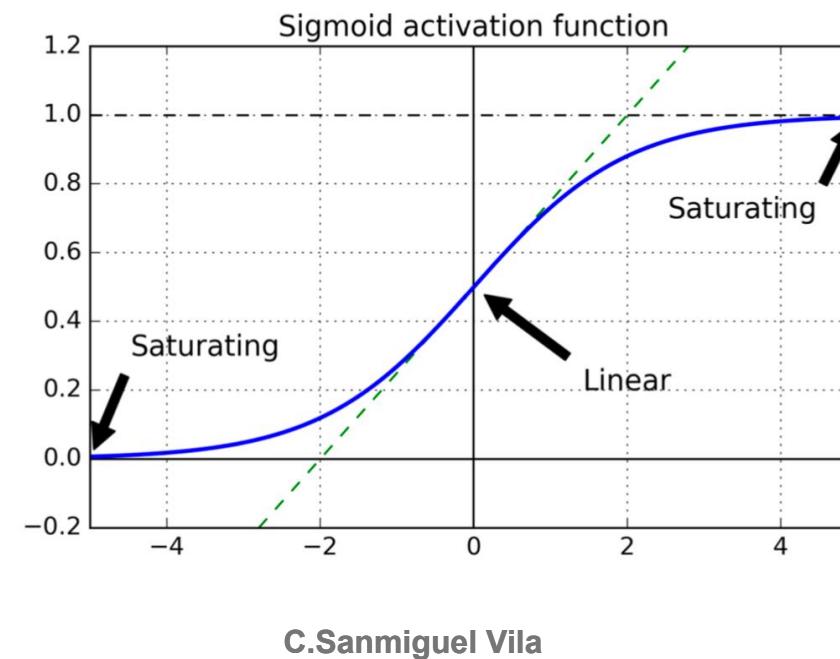
- **Vanishing/exploding gradients**
 - Gradients grow smaller and smaller or larger and larger as we flow backwards through the network during training
- **Regularization**
 - We might not have enough training data for training a large network or data instances are too noisy
 - Batch normalization and dropout
 - L_1 and L_2 regularization
- **Optimizers**
 - Various optimization methods can speed up training large neural networks

Vanishing/exploding gradients problem

- When gradients get smaller and smaller, the Gradient Descent update leaves many connection weights unchanged
 - **Known as the vanishing gradients problem**
- When gradients grow bigger and bigger, some layers get large weight updates and the algorithm diverges
 - **Known as the exploding gradients problem**
- One of the main reasons deep neural networks were abandoned in 2000s
- It appears there are two main factors
 - Weight initialization
 - Sigmoid activation function

Activation function saturation

- When inputs become large (negative or positive), the function saturates at 0 or 1
- Derivative extremely close to 0
- No gradient to propagate back through the network



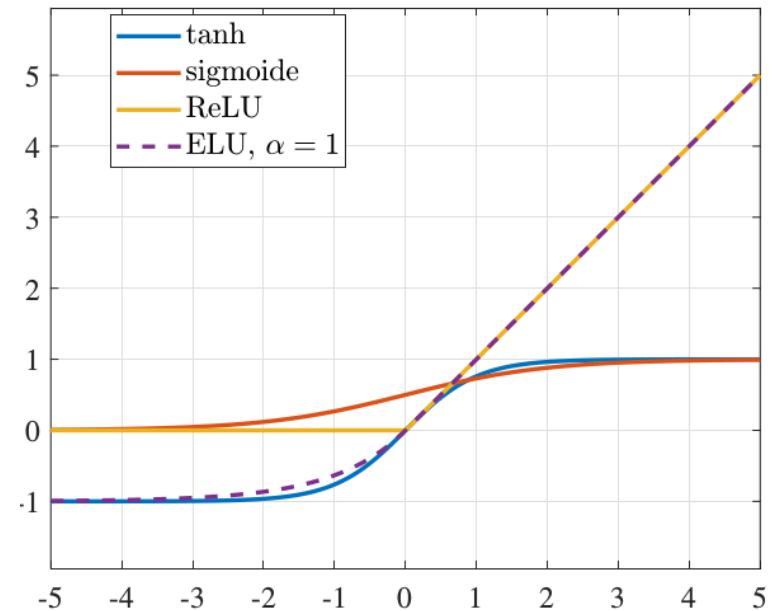
Why Does Weight Initialization Matter?

The main issues with poor weight initialization are:

- **Vanishing gradients:** If weights are initialized too small, they can become exponentially smaller during backpropagation as they move backwards through the layers. This causes the gradients to “vanish,” resulting in extremely slow learning or even complete stagnation.
- **Exploding gradients:** If weights are initialized too large, the gradients can become very large during backpropagation, which may cause the model’s weights to grow uncontrollably, leading to divergence in training.
- **Symmetry breaking:** Initializing all weights to the same value (e.g., 0) would cause neurons to learn the same features, effectively making the neurons redundant. Random initialization is used to prevent this.

Nonsaturating activation functions

- We know that ReLU activation function does not saturate for positive values, and it is fast to compute
- However, ReLU is not perfect!
- Dying ReLU problem: some neurons effectively “die,” meaning they stop producing anything other than 0
- Solution: $\text{ELU}(x) = x$ if $x > 0$, else $\alpha(e^x - 1)$ with $0 < \alpha$
- If $\alpha = 1$, the function is smooth everywhere including $x = 0$



Glorot and He Initialization

- **Goal:** the variance of the outputs of each layer to be equal to the variance of its inputs
- Number of inputs: fan_in
- Number of output neurons: fan_out
- Let us define $\text{fan_avg} = (\text{fan_in} + \text{fan_out})/2$

Equation 11-1. Glorot initialization (when using the logistic activation function)

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

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

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / \text{fan}_{\text{avg}}$
He	ReLU and variants	$2 / \text{fan}_{\text{in}}$
LeCun	SELU	$1 / \text{fan}_{\text{in}}$

Glorot and He Initialization

- **Xavier (Glorot) Initialization:** Designed for activation functions like sigmoid and tanh, where it attempts to keep the variance of the inputs and outputs consistent across layers. The weights are drawn from a distribution with a variance based on the number of input and output units.
- **He Initialization (Kaiming Initialization):** Developed for ReLU-like activation functions, which "kill" negative activations by setting them to zero. He initialization accounts for the asymmetry of ReLU. It sets the weights to a distribution that helps maintain a good flow of gradients during training.
- By default, PyTorch uses a form of Xavier (Glorot) uniform initialization for the weights in layers like nn.Linear and nn.Conv2d, with biases initialized to zeros.

Glorot and He Initialization

```
import torch.nn.init as init

# Manually initialize weights using Xavier uniform
def init_weights(m):
    if isinstance(m, nn.Linear):
        init.xavier_uniform_(m.weight)
        init.zeros_(m.bias)

model = SimpleModel()
model.apply(init_weights) # Apply custom initialization
```

python

Copiar código

```
def init_weights(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
        init.kaiming_uniform_(m.weight, nonlinearity='relu') # He initialization
        if m.bias is not None:
            init.zeros_(m.bias) # Set biases to zero

model.apply(init_weights)
```

Choosing the Right Initialization

- Xavier Initialization is typically used for sigmoid and tanh activation functions, where the gradients can vanish.
- He Initialization is preferred for ReLU and its variants, as it helps maintain gradient flow by initializing weights with larger values.
- ELU can also benefit from He Initialization, but some experimentation might be necessary.
- If you're unsure, PyTorch's default initialization (Xavier) is generally a good starting point for most networks, and you can experiment with custom initialization if necessary.

Batch normalization

- Batch normalization is a regularization technique, although it primarily aims to stabilize and speed up training. It works by normalizing the outputs of each layer to have a mean of 0 and a variance of 1 across the batch. This reduces internal covariate shift, making the model's convergence easier.
- For each batch, batch normalization normalizes the output h of a layer:

$$\hat{h} = \frac{h - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where

- μ_B is the mean of the batch,
- σ_B^2 is the variance of the batch,
- ϵ is a small constant for numerical stability.
- Batch normalization also introduces two trainable parameters γ and β , which allow the network to learn the optimal scale and shift for the normalized outputs.

Equation 11-3. Batch Normalization algorithm

$$1. \quad \mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$2. \quad \sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$$

$$3. \quad \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$4. \quad \mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$$

Batch normalization

```
class MLPWithBatchNorm(nn.Module):
    def __init__(self):
        super(MLPWithBatchNorm, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.bn1 = nn.BatchNorm1d(512) # BatchNorm after first fully connected layer
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256) # BatchNorm after second fully connected layer
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x))) # Apply batch norm followed by ReLU
        x = F.relu(self.bn2(self.fc2(x))) # Apply batch norm followed by ReLU
        x = self.fc3(x)
        return x

# Example of usage
model = MLPWithBatchNorm()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Batch normalization

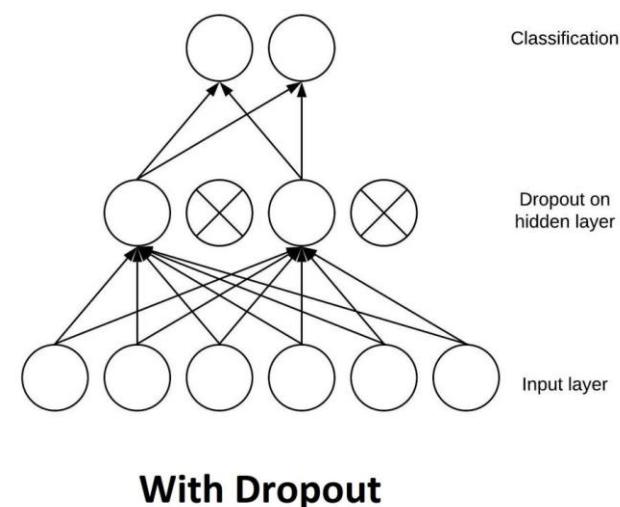
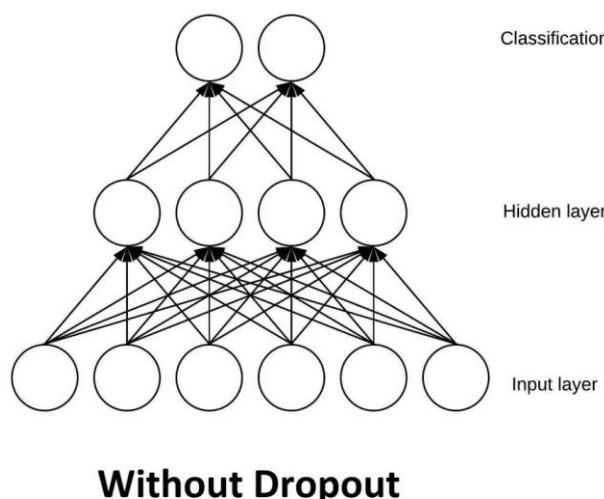
In this model, we apply batch normalization after the linear transformations and before the activation function (ReLU). This ensures that the inputs to the next layer have stable distributions, making the model easier to train.

Effects of Batch Normalization:

- **Faster Convergence:** By normalizing the outputs, the network can converge faster during training
- **Smoothing the Loss Landscape:** Batch normalization reduces the problem of vanishing/exploding gradients
- **Regularization Effect:** Batch normalization has a mild regularization effect, as each mini-batch's statistics are slightly different

Dropout

- Dropout is a simple but effective regularization technique. It works by randomly "dropping" or setting a fraction of neurons to zero during training, which prevents the network from becoming too reliant on specific neurons and encourages it to learn more robust, distributed representations
- If a layer has a set of neurons $h=\{h_1, h_2, \dots, h_n\}$, during training, Dropout randomly sets some neurons h_i to zero with a probability p . The remaining neurons are scaled by $\frac{1}{1-p}$ to maintain the overall contribution to the next layer.



Dropout

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class MLPWithDropout(nn.Module):
    def __init__(self):
        super(MLPWithDropout, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 10)
        self.dropout = nn.Dropout(p=0.5) # Dropout with probability 0.5

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x) # Apply dropout after first layer
        x = F.relu(self.fc2(x))
        x = self.dropout(x) # Apply dropout after second layer
        x = self.fc3(x)
        return x

# Example of usage
model = MLPWithDropout()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Copiar código

Dropout

Dropout helps reduce overfitting by forcing the network to learn more robust features. It can slow down convergence since the network is effectively training different subnetworks each time, but the final model tends to generalize better.

Combining Dropout and Batch Normalization:

- You can combine both techniques in a network, but typically **dropout** is applied after fully connected layers, while **batch normalization** is applied between the linear layer and the activation function.

Key Takeaways:

- **Dropout:** Prevents overfitting by randomly dropping neurons during training. It is typically used after fully connected layers in both MLPs and CNNs.
- **Batch Normalization:** Speeds up training by normalizing the output of each layer to reduce internal covariate shift. It can be used after linear layers (or convolutional layers for CNNs) and before activation functions.

Dropout

```
class MLPWithDropoutAndBatchNorm(nn.Module):
    def __init__(self):
        super(MLPWithDropoutAndBatchNorm, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.bn1 = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 10)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

# Example of usage
model = MLPWithDropoutAndBatchNorm()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

L1 and L2 Regularization

L1 and L2 regularization are two of the most used techniques for adding constraints to model weights during training to reduce overfitting. They are also known as weight regularization because they penalize the size of the model's weights. By introducing a penalty term to the loss function, these methods discourage the model from learning overly complex patterns that may not generalize well to unseen data.

L2 Regularization (Ridge Regularization)

- L2 regularization adds a penalty proportional to the **square** of the magnitude of the weights to the loss function. This helps prevent the weights from becoming too large, which can help generalize new data.

L1 Regularization (Lasso Regularization)

- L1 regularization adds a penalty proportional to the **absolute value** of the weights to the loss function. This can lead to sparse models where many weights are driven to exactly zero, effectively performing feature selection by removing irrelevant features.

L1 and L2 Regularization

Key Differences Between L1 and L2 Regularization

- L1 Regularization: Drives some weights to exactly zero, leading to a sparse model that can act as feature selection. This is particularly useful in high-dimensional datasets where many features may be irrelevant.
- L2 Regularization: Shrinks weights uniformly but does not necessarily drive them to zero. It helps distribute the impact of different features more smoothly and can lead to more balanced models.

When to Use L1 or L2

- L1: Use L1 regularization when you suspect that some features are irrelevant, and you want the model to automatically select a subset of important features. This is especially useful in high-dimensional feature spaces, like text processing or genomics.
- L2: Use L2 regularization when you want to prevent large weights but don't expect a large number of irrelevant features. It's useful when all features are expected to contribute to the output but may overfit without regularization.

L1 and L2 Regularization

In PyTorch, L2 regularization is often applied through the `weight_decay` parameter in optimizers such as Adam, SGD, etc.

```
# Use L2 regularization by setting the weight_decay parameter
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01) # L2 regularization
```

L1 regularization

```
# Define a function for L1 regularization
def l1_regularization(model, lambda_l1):
    l1_norm = sum(param.abs().sum() for param in model.parameters())
    return lambda_l1 * l1_norm

# Example training loop with L1 regularization
lambda_l1 = 0.001 # Regularization strength for L1

for epoch in range(epochs):
    for batch in train_loader:
        data, target = batch
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target)

        # Add L1 regularization to the loss
        loss += l1_regularization(model, lambda_l1)

        loss.backward()
        optimizer.step()
```

Popular Optimizers in PyTorch

In practice, several optimizers build on top of basic gradient descent to make training more efficient, faster, and stable.

Stochastic Gradient Descent (SGD)

SGD is one of the most straightforward optimization techniques. It updates weights using the gradient of a small batch of data, which can make updates noisy but faster.

$$W = W - \eta \nabla_W L$$

```
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Drawback: SGD can oscillate a lot near minima because it uses raw gradients. Therefore, it's often combined with techniques like **momentum**.

Popular Optimizers in PyTorch

SGD with Momentum

Momentum is a technique that helps SGD overcome the problem of oscillation, especially in regions where gradients vary greatly in magnitude. It does this by maintaining an exponentially decaying moving average of past gradients.

$$\begin{aligned}v_t &= \beta v_{t-1} + (1 - \beta) \nabla_W L \\W &= W - \eta v_t\end{aligned}$$

Where β is the momentum coefficient (usually between 0.9 and 0.99).

Momentum helps accelerate convergence, particularly in areas where gradients are noisy or highly varied.

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Popular Optimizers in PyTorch

RMSProp (Root Mean Square Propagation)

RMSProp adapts the learning rate based on the average of recent squared gradients. This prevents the learning rate from becoming too large in directions where gradients are large.

$$\begin{aligned}v_t &= \beta v_{t-1} + (1 - \beta)(\nabla_W L)^2 \\W &= W - \frac{\eta}{\sqrt{v_t} + \epsilon} \nabla_W L\end{aligned}$$

```
optimizer = optim.RMSprop(model.parameters(), lr=0.001)
```

Popular Optimizers in PyTorch

Adam (Adaptive Moment Estimation)

Adam is one of the most widely used optimizers because it combines the best of both worlds: momentum and adaptive learning rates. It typically requires less tuning than SGD. Adam maintains both:

- An exponentially decaying average of past gradients (similar to momentum).
- An exponentially decaying average of past squared gradients, which helps adapt the learning rate for each parameter individually.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_W L; v_t = \beta v_{t-1} + (1 - \beta) \nabla_W^2 L$$
$$\widehat{m}_t = \frac{m_t}{1 - \beta_1}; \widehat{v}_t = \frac{v_t}{1 - \beta_2}; W = W - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$$

Where:

- \widehat{m}_t is the first moment estimate (momentum-like term).
- \widehat{v}_t is the second moment estimate (RMSProp-like term).
- ϵ is a small constant to prevent division by zero.

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Learning Rate Scheduling

The learning rate is one of the most critical hyperparameters in optimization. A high learning rate can lead to divergence, while a low learning rate can result in slow convergence. Therefore, learning rate scheduling is often used to adjust the learning rate dynamically during training.

Common strategies include:

- **Step Decay:** Reduce the learning rate by a factor after a certain number of epochs.

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
```

This will reduce the learning rate by a factor of 0.1 every 30 epochs.

- **Reduce on Plateau:** Reduce the learning rate if the validation loss stops improving.

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=10, factor=0.1)
```

This reduces the learning rate if the loss plateaus.

- **Cosine Annealing:** The learning rate is adjusted following a cosine function over the training epochs, which can help the optimizer converge more smoothly.

```
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=50)
```

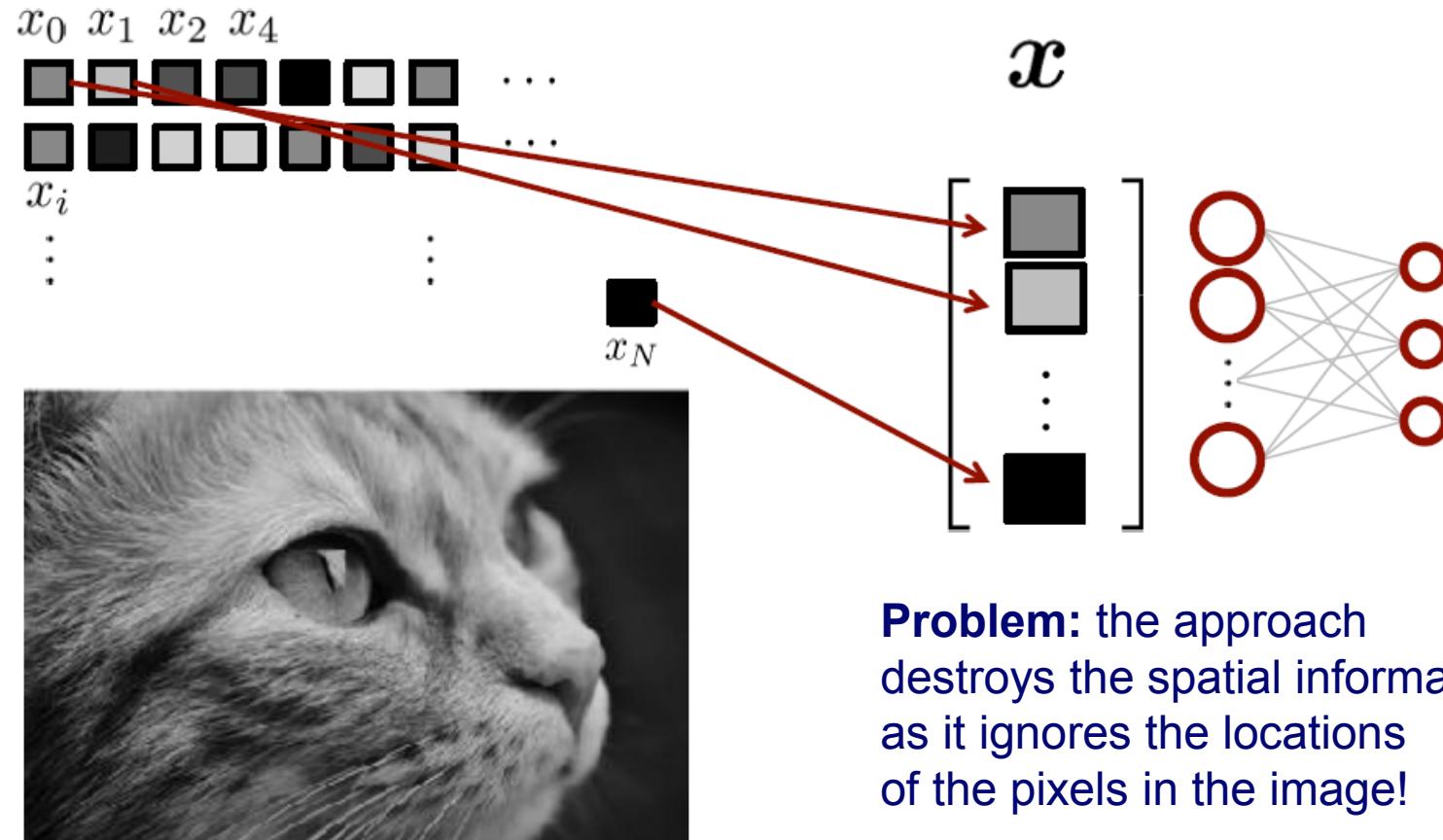
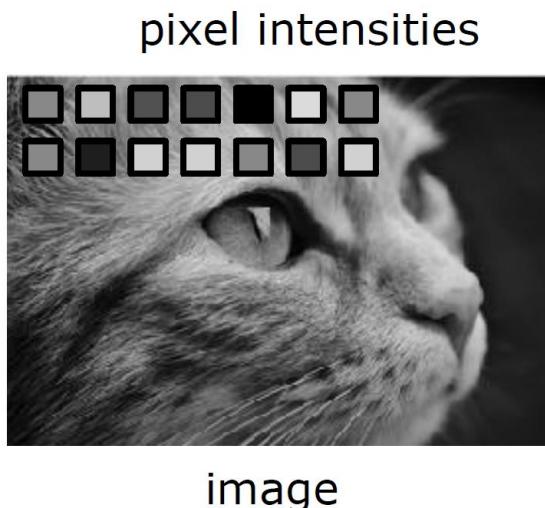
Data-intensive space engineering

Lecture 9

Carlos Sanmiguel Vila

Based on previous work of Cyrill
Stachniss from University of
Bonn

The Good Old MLP's Input...



Problem: the approach destroys the spatial information as it ignores the locations of the pixels in the image!

CNNs Overcome this Problem

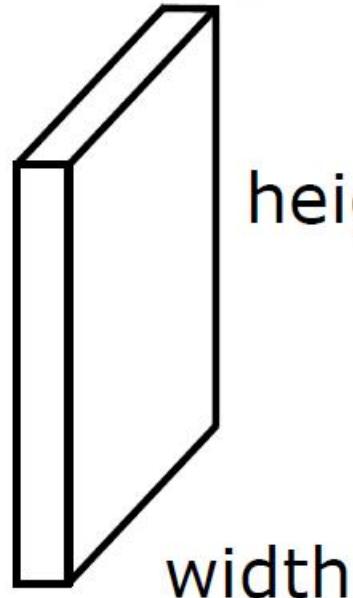
- CNNs maintain the 2D image structure
- Neighborhoods are maintained
- Network layers can learn features that also encode spatial information
- Convolutions are local operators
- CNNs use convolutions & subsampling (called pooling)
- Thanks to the increase in computational power and the amount of available training data, convolutional neural networks (CNNs) have achieved great performance on complex visual tasks
 - Image search services, self-driving cars, video classification systems, etc.
 - Not restricted to visual applications, e.g., voice recognition

CNNs Overcome this Problem

- CNNs maintain the 2D image structure
- Neighborhoods are maintained
- Network layers can learn features that also encode spatial information
- Convolutions are local operators
- CNNs use convolutions & subsampling (called pooling)
- Thanks to the increase in computational power and the amount of available training data, convolutional neural networks (CNNs) have achieved great performance on complex visual tasks
 - Image search services, self-driving cars, video classification systems, etc.
 - Not restricted to visual applications, e.g., voice recognition

Let's Start With the Input

channels/depth

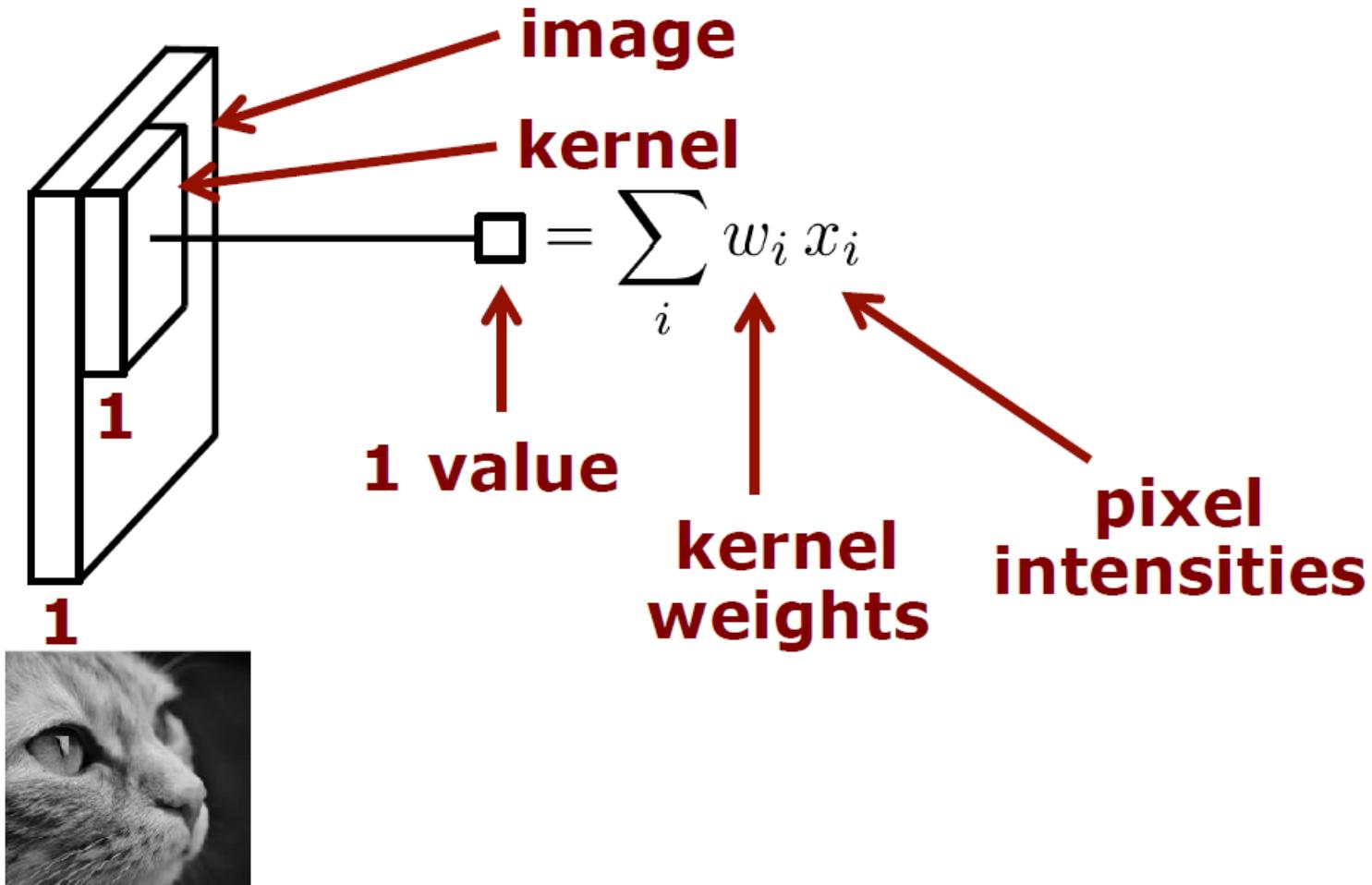


depth=1

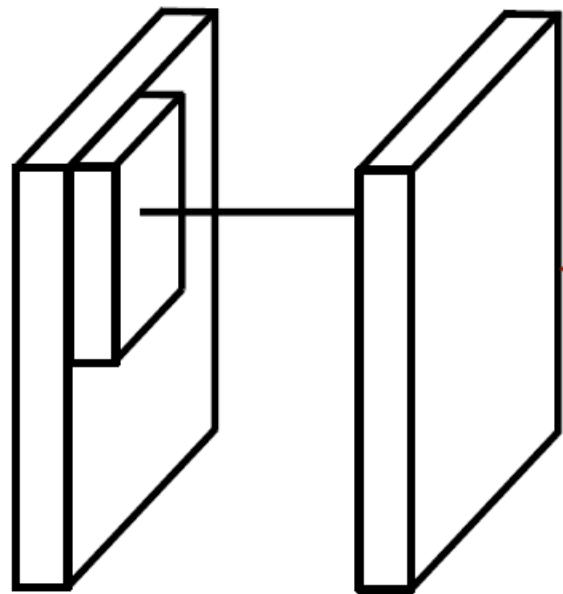


depth=3

Convolution Using a Kernel



Convolution Using a Kernel

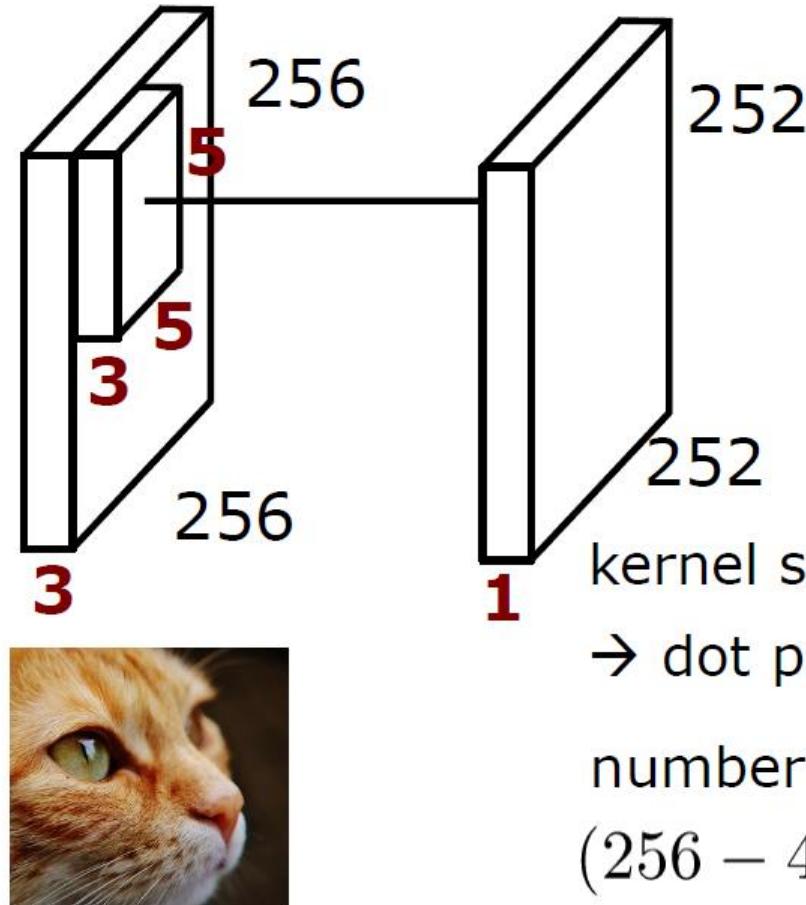


**This is the
output (image)
of a convolution!**



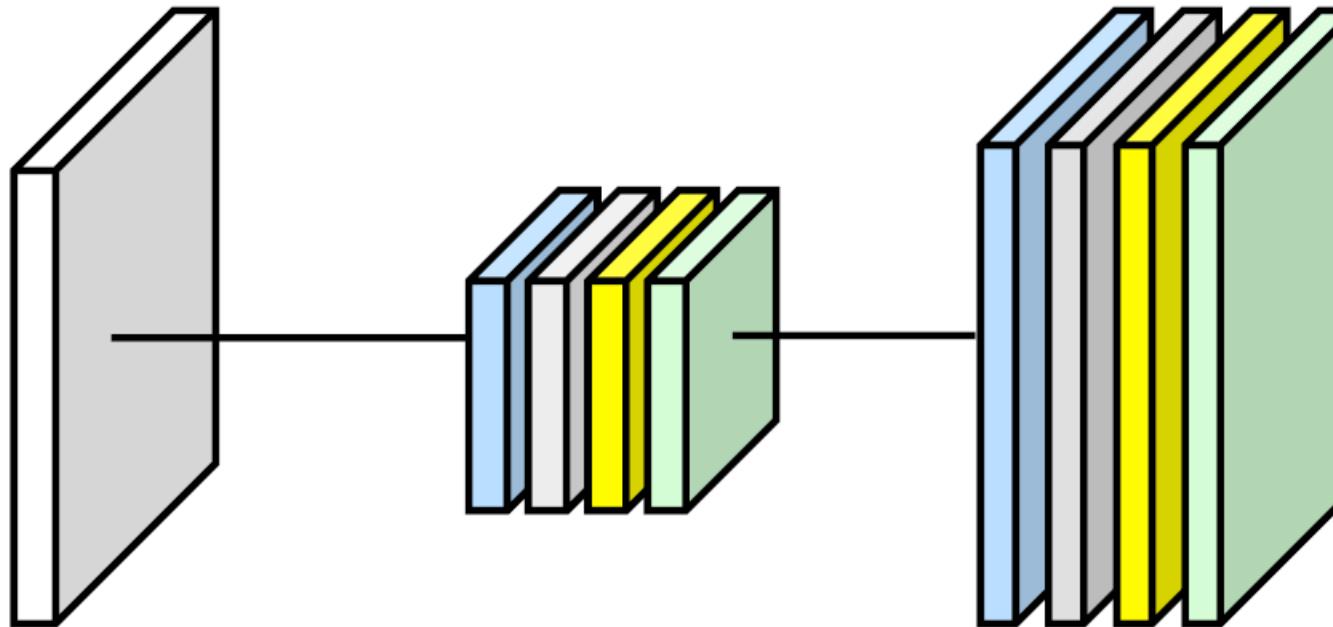
example for
blurring through
a convolution

Convolution Using a Kernel



kernel size: $3 \times 5 \times 5 = 75$
→ dot product of 75 dim. vectors
number of such dot products:
 $(256 - 4) \times (256 - 4) = 63.5k$

Convolution Using Multiple Kernel



1 input

$3 \times W \times H$

4 kernels

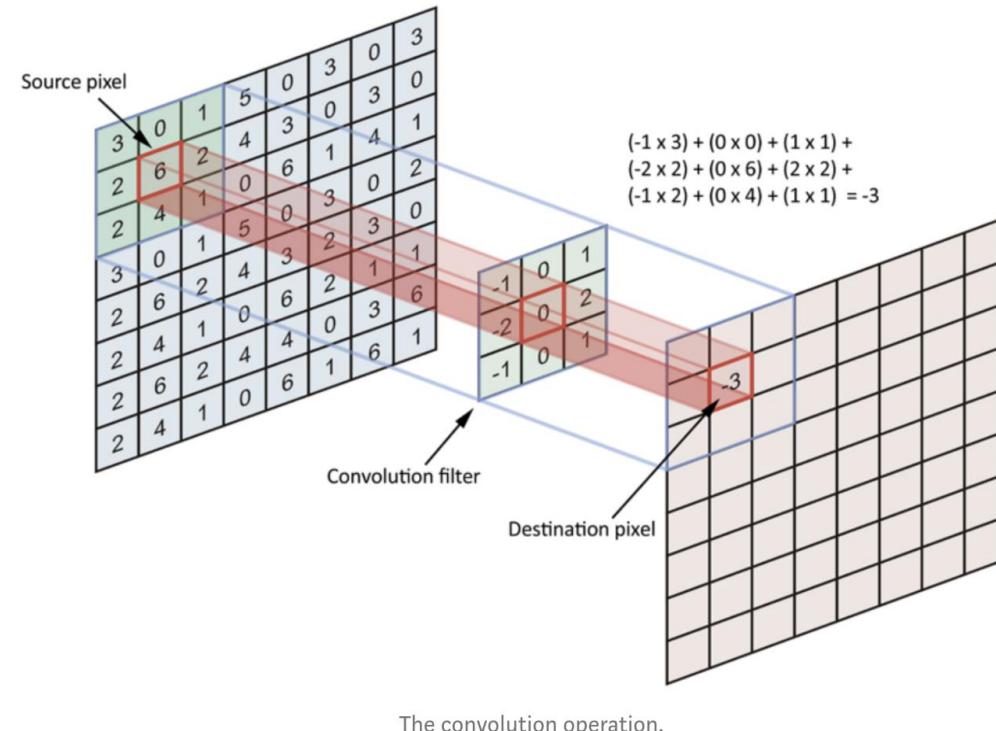
$4 \times 3 \times 5 \times 5$

4 outputs
activation maps

$4 \times 1 \times (W-4) \times (H-4)$

Convolution Kernels

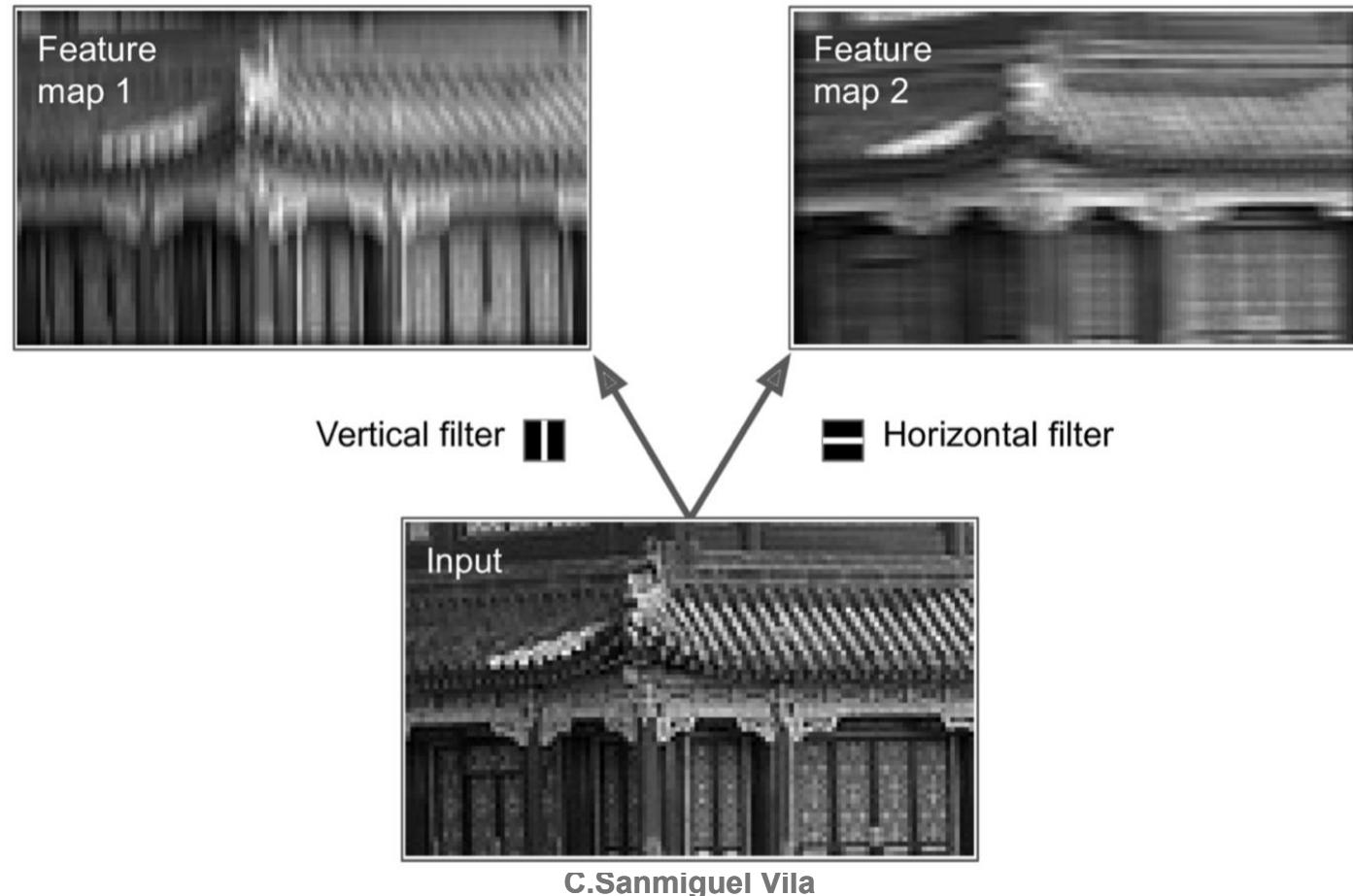
- A neuron's weight can be represented as a small image of the size of the receptive field
- We refer to sets of weights as filters or convolutional kernels



The convolution operation.

Convolution Kernels

- Let us consider two possible sets of weights, i.e., filters
- We obtain feature maps, highlighting the areas in an image that activate the filter the most



Output Sizes

Size of the activation map depends on

- Size of the input (Width, Height)
- Kernel size (K)

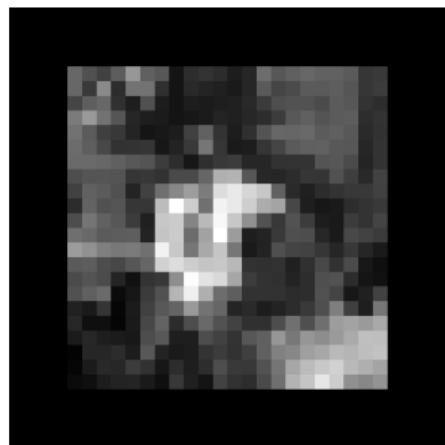
$$W' \times H' = (W - K + 1) \times (H - K + 1)$$

Output size

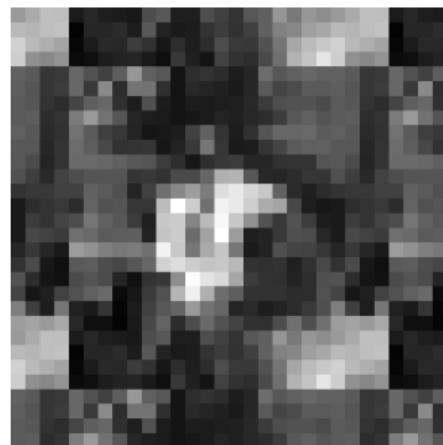
Input size and Kernel size

Padding

- Convolutions slightly shrink the image
- We can solve this by creating a border around the input image



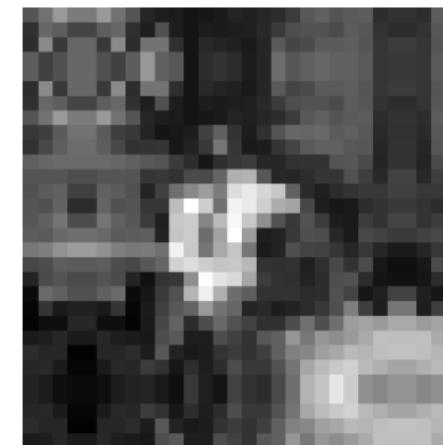
zero



wrap



clamp



mirror

CNNs often use zero-padding

Image courtesy: Szelinsky

Output Sizes with Padding

Size of the activation map depends on

- Size of the input (Width, Height)
- Kernel size (K)
- Padding (P)

$$W' \times H' = (W - K + 1 + 2P) \times (H - K + 1 + 2P)$$

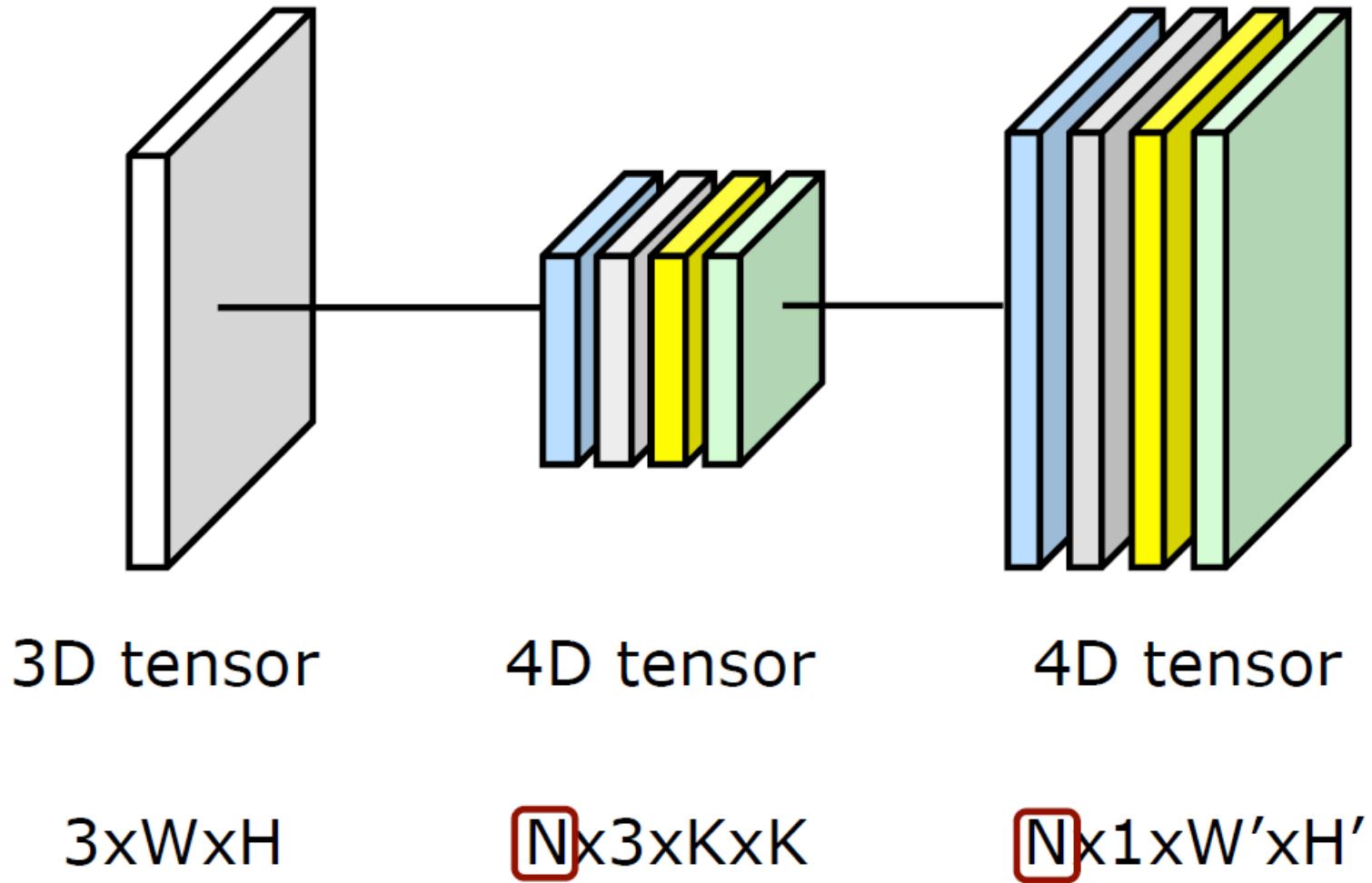
Tensor

- Vector is a 1-dimensional array
 - Matrix is a 2-dimensional array
 - Voxelgrid is a 3-dimensional array

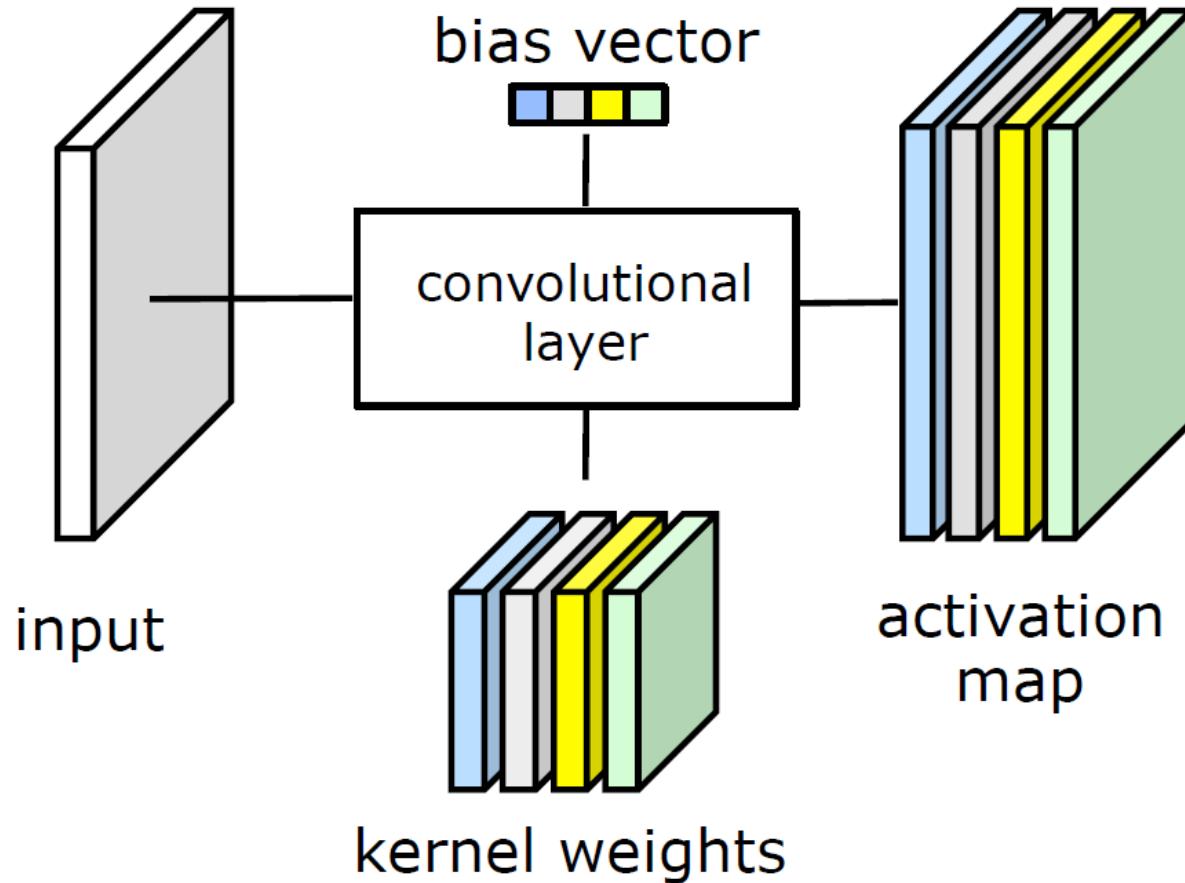
$$\epsilon_{ijk} =$$

[Image courtesy: A. Kriesch]

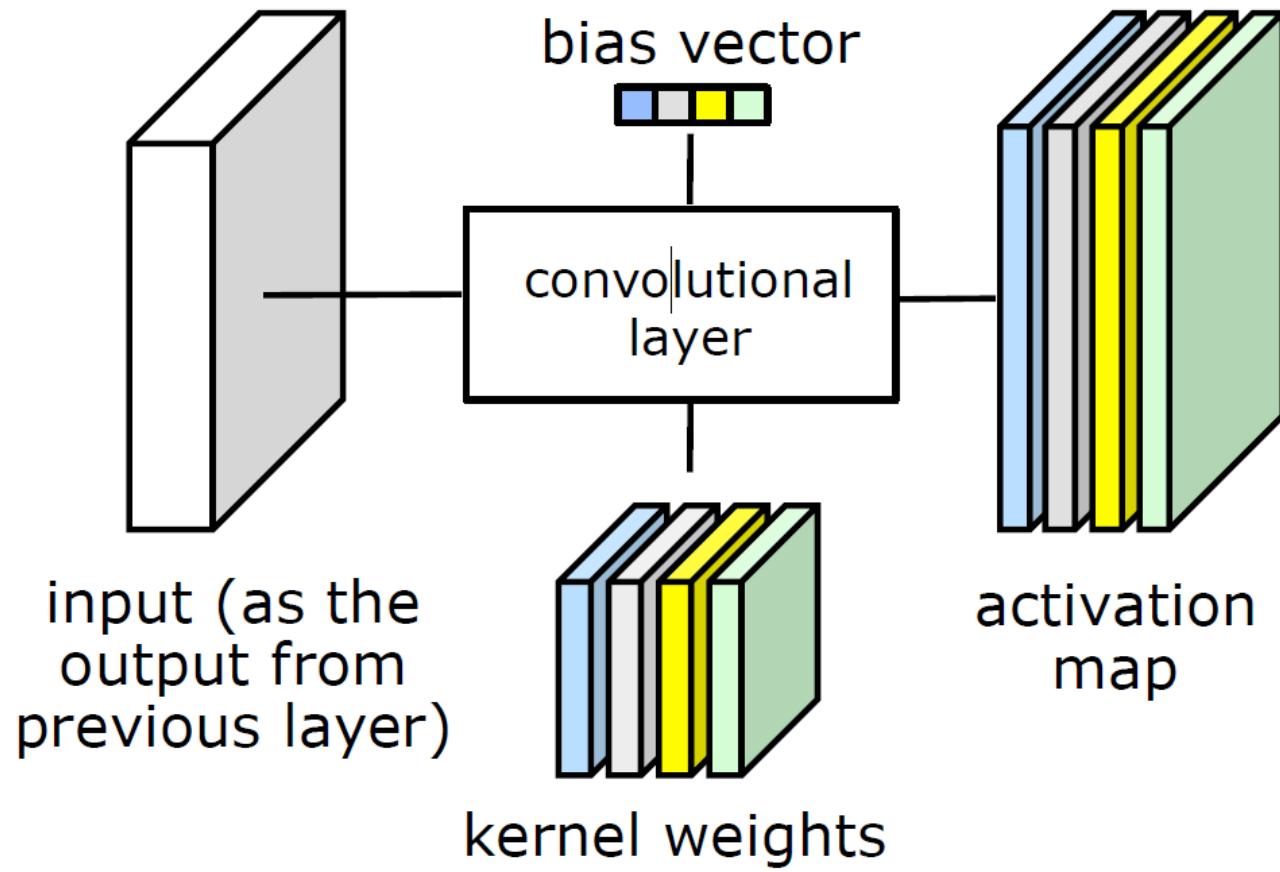
Each Layer is a Tensor



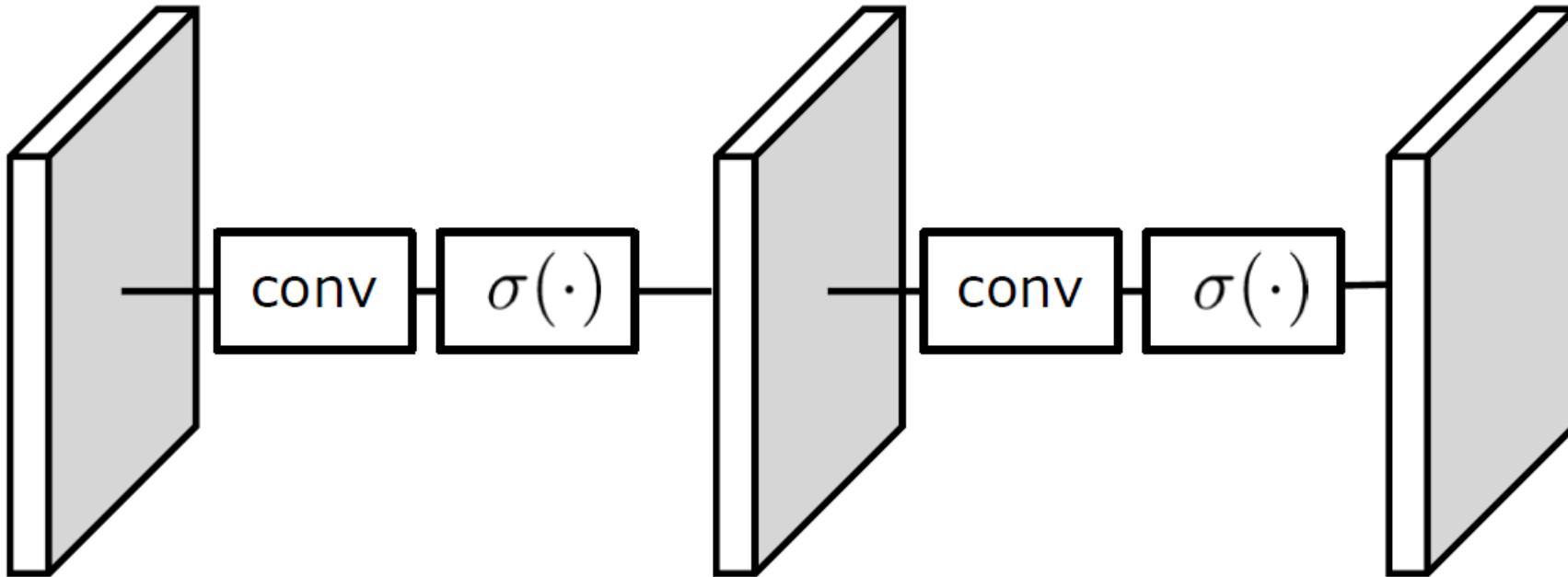
Convolutional Layer Parameters



Stacking Convolutional Layers

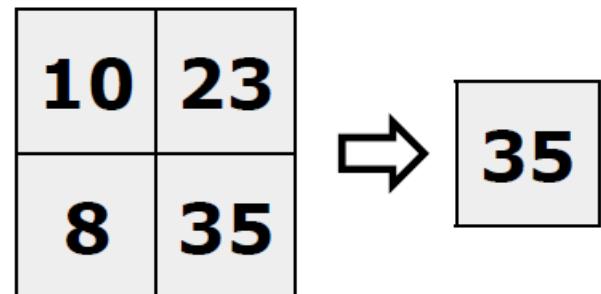


Stacking Convolutions Layers With Activation Functions

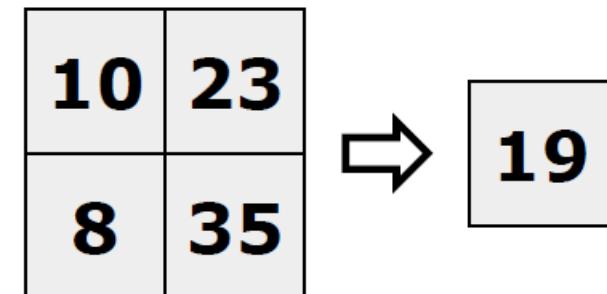


Pooling

- Besides convolutions, CNNs also use pooling layers
- Pooling combines multiple values into a single value to reduce the tensor sizes and combine information
- We want to reduce the computational load and memory usage
- Prominent examples are:



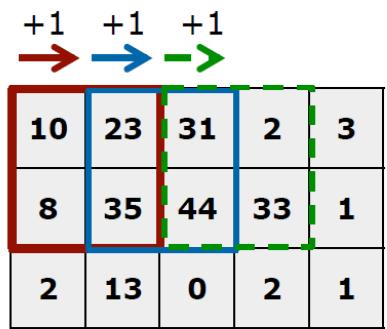
max-pooling



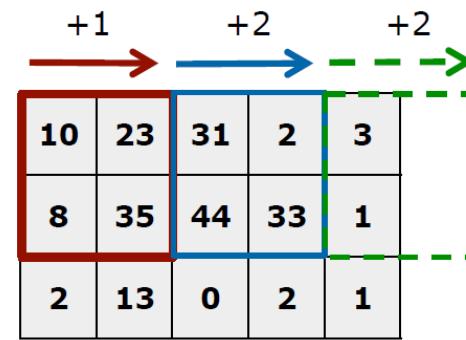
avg-pooling

Stride

- Stride defines by how many pixels we shift the filter forward each step
- Larger stride reduces overlaps and makes the resulting image smaller



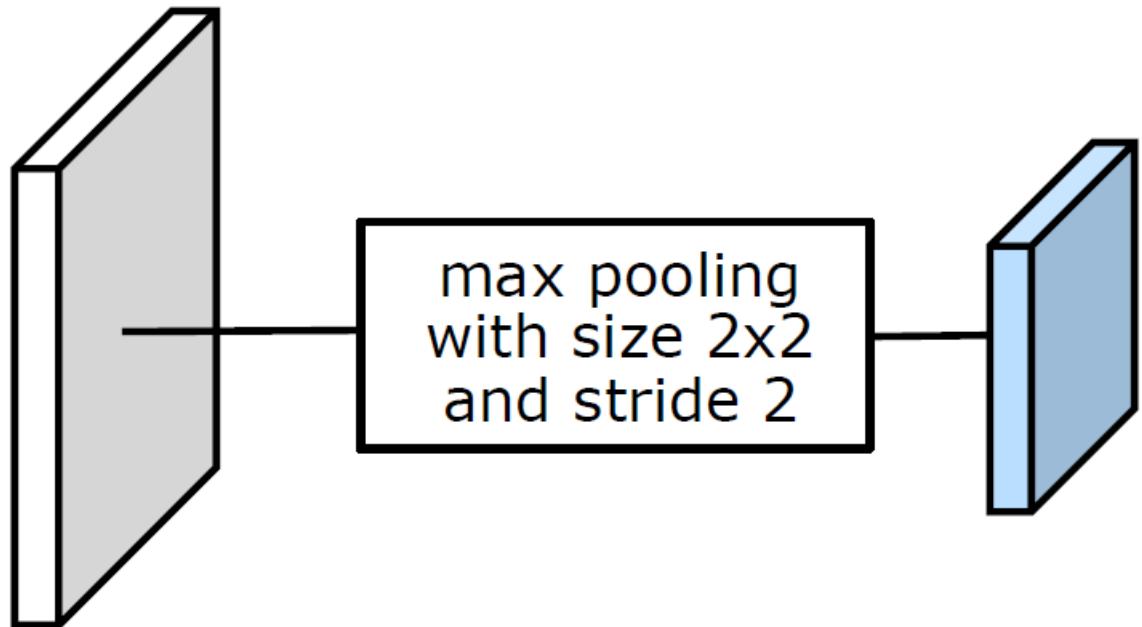
stride=1



stride=2

Max Pooling Example

Size tells us how many values to combine, and stride define by how much to shift the mask

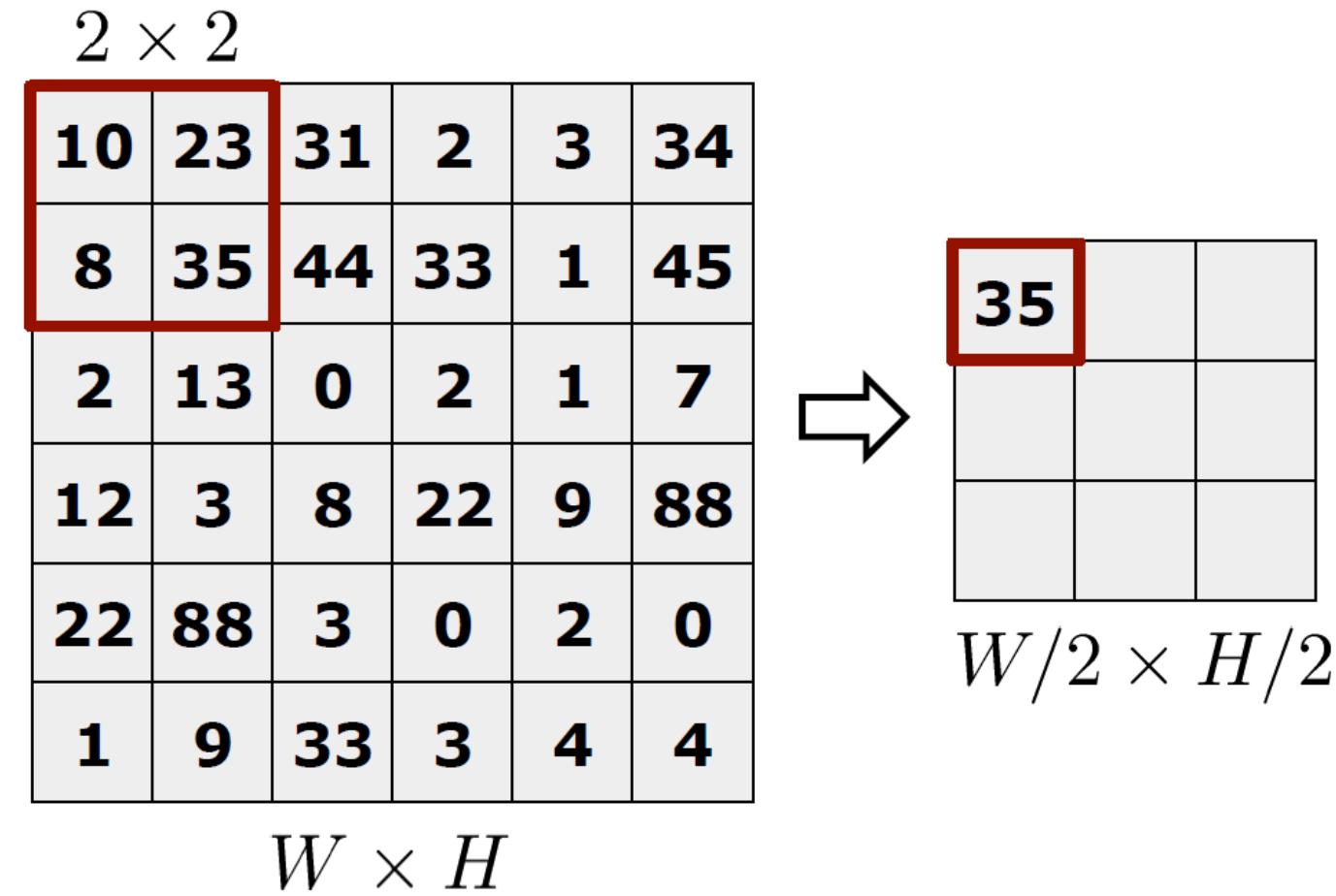


$$W \times H$$

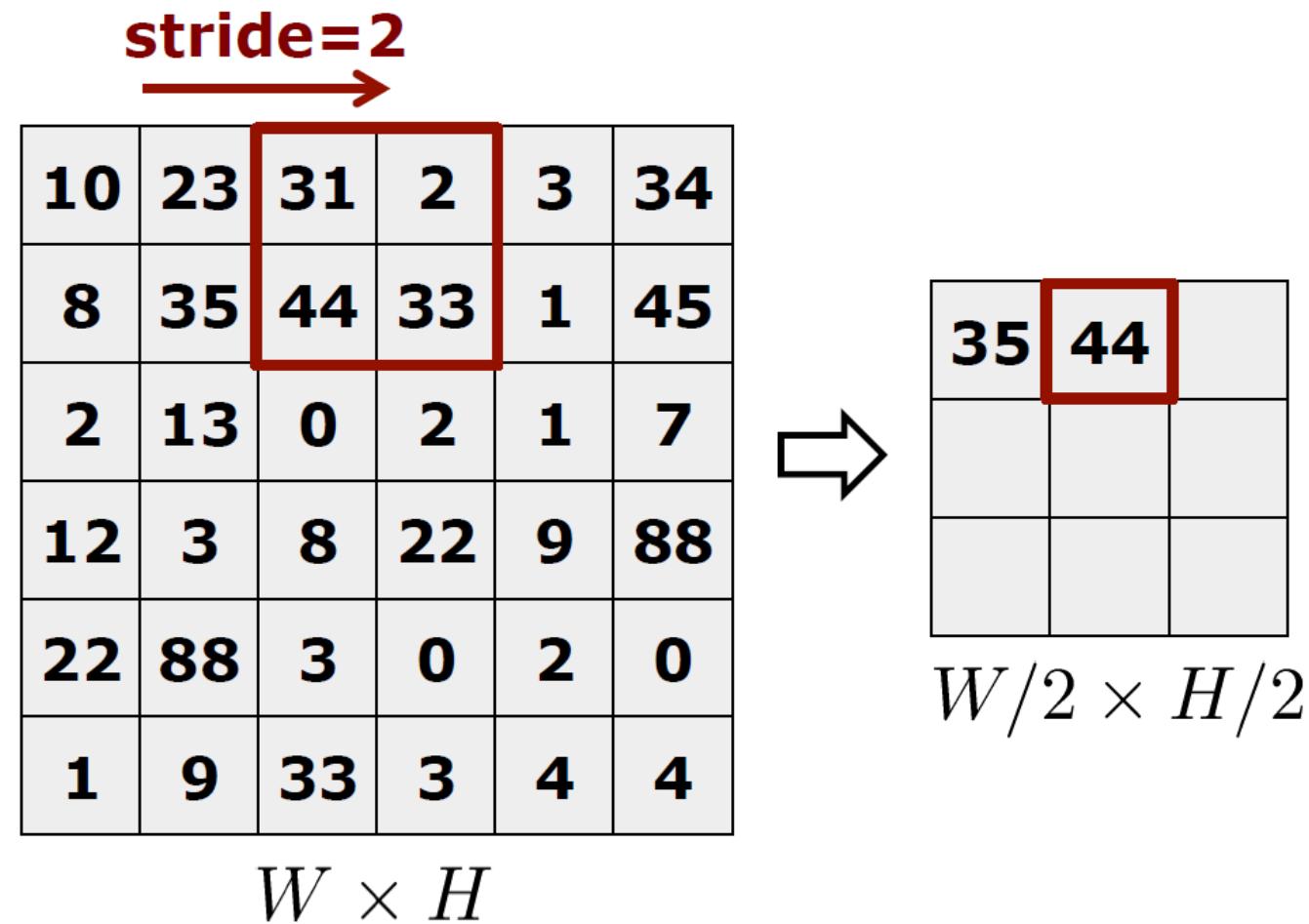
$$2 \times 2$$

$$W/2 \times H/2$$

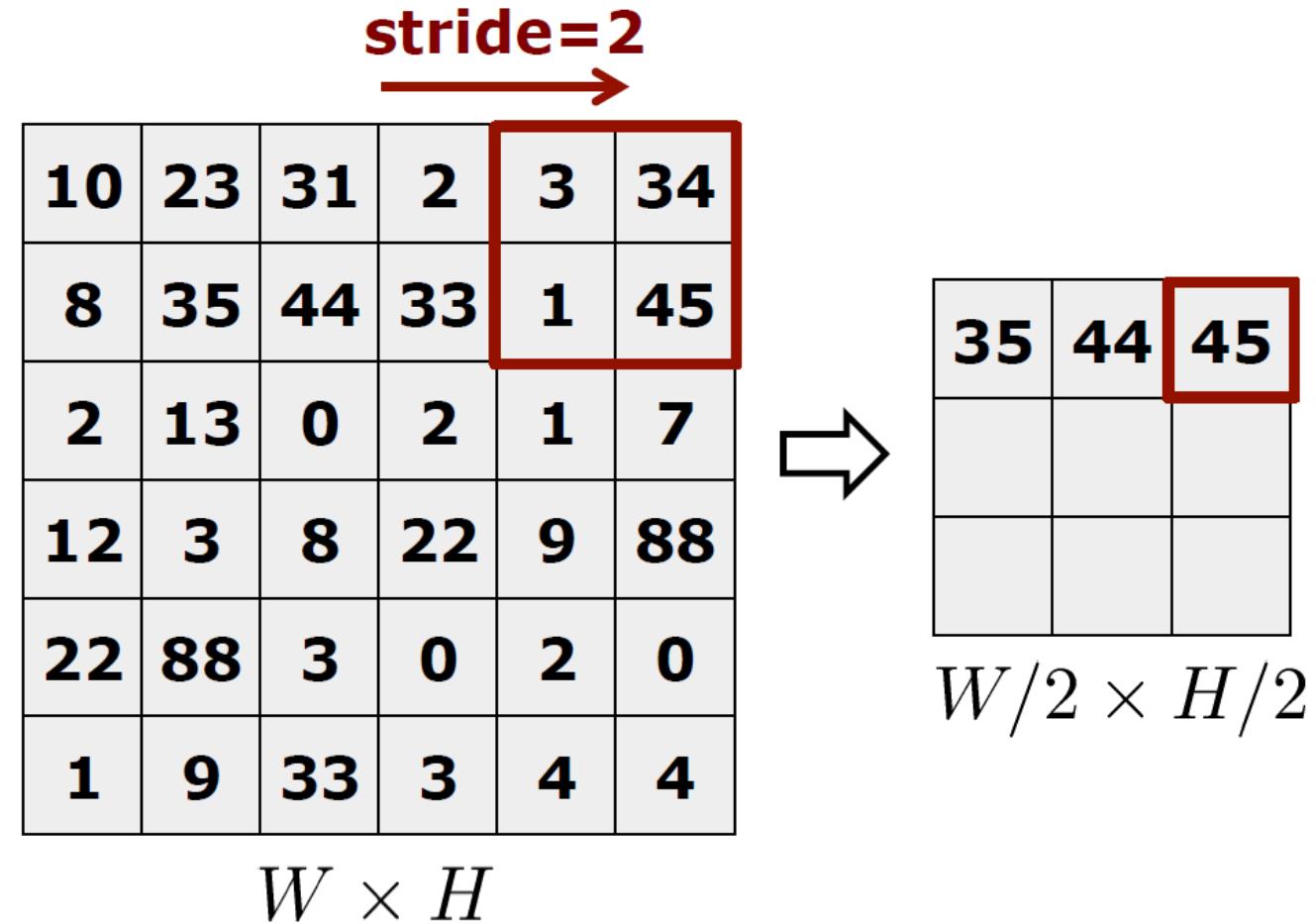
Max Pooling Example



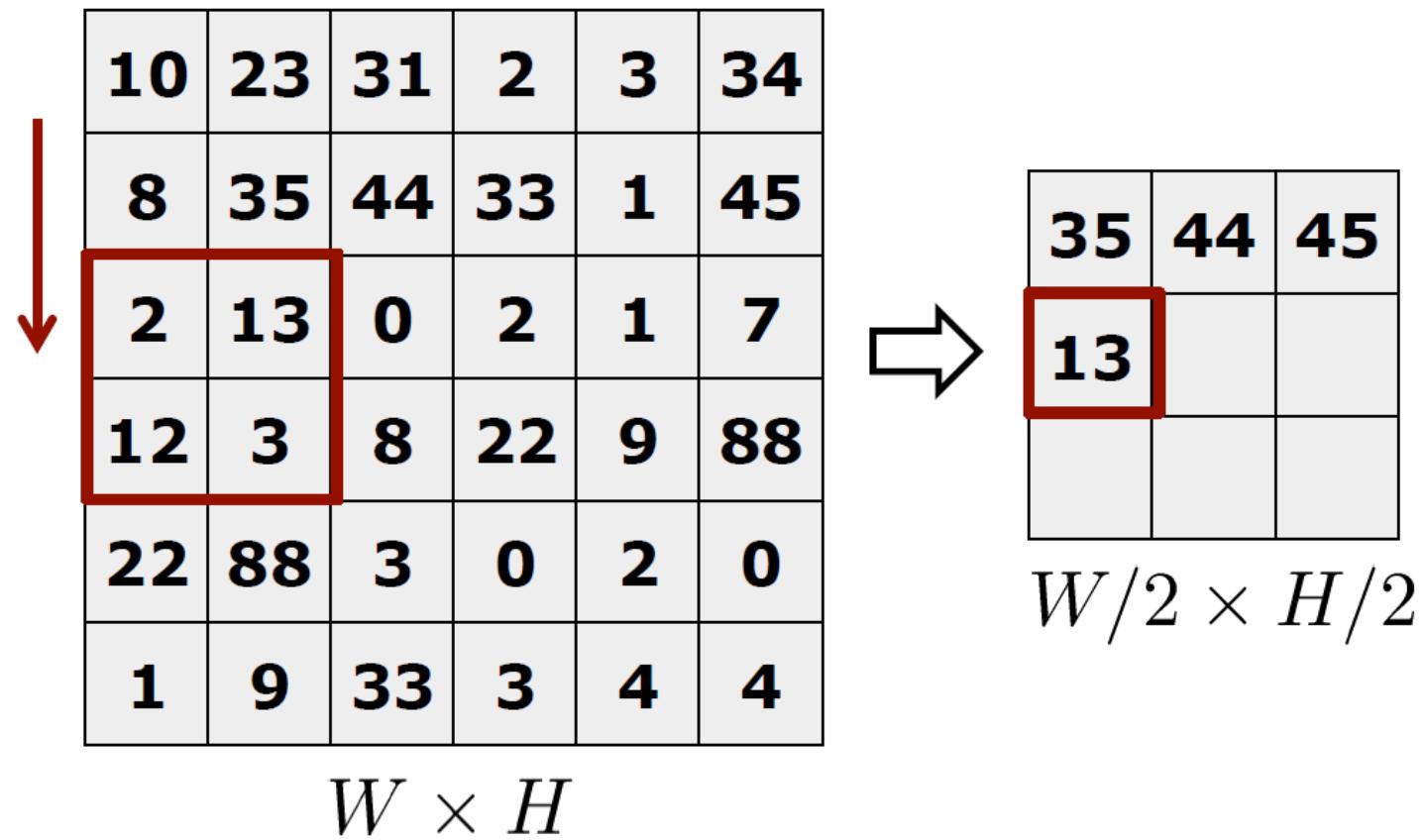
Max Pooling Example



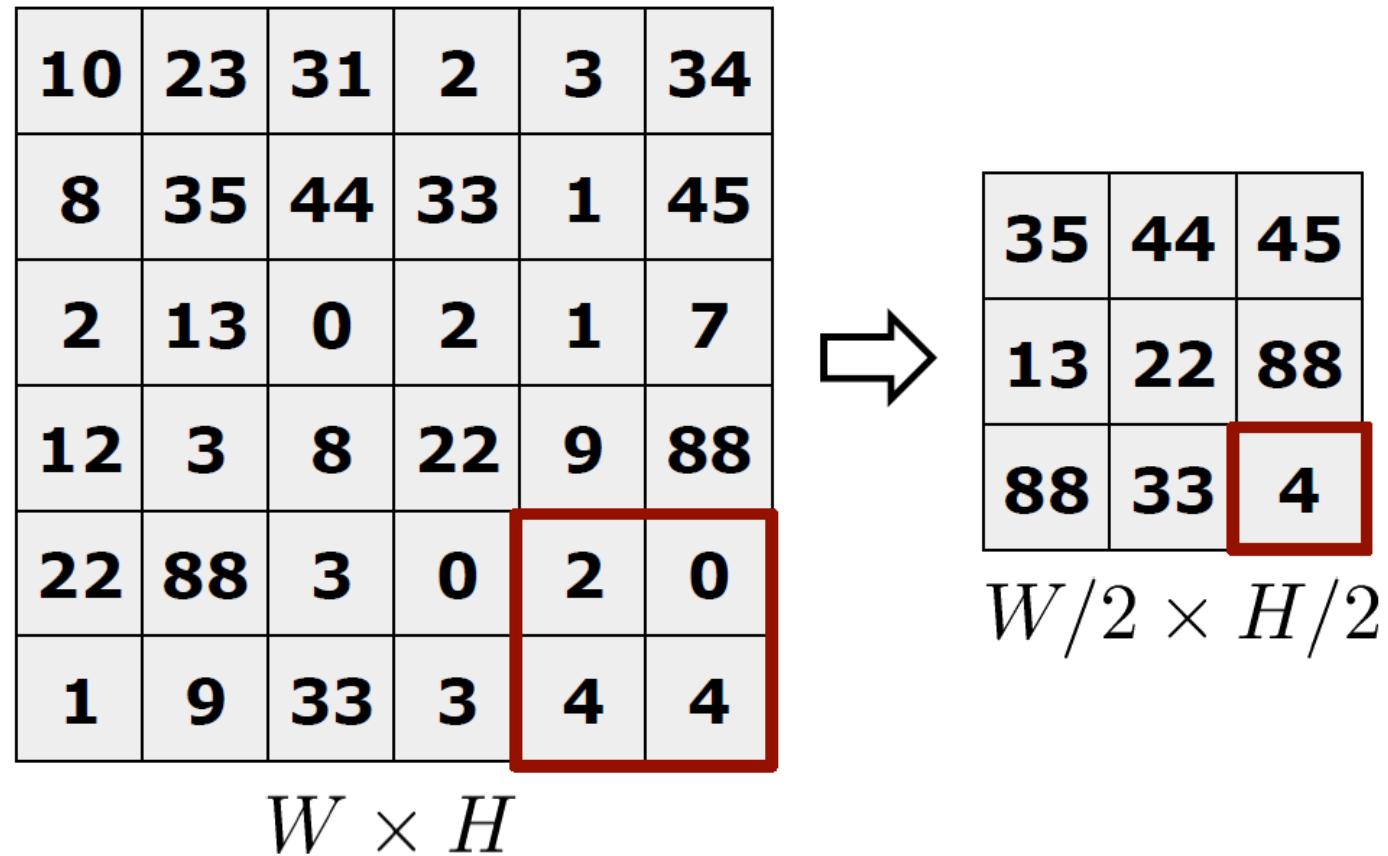
Max Pooling Example



Max Pooling Example

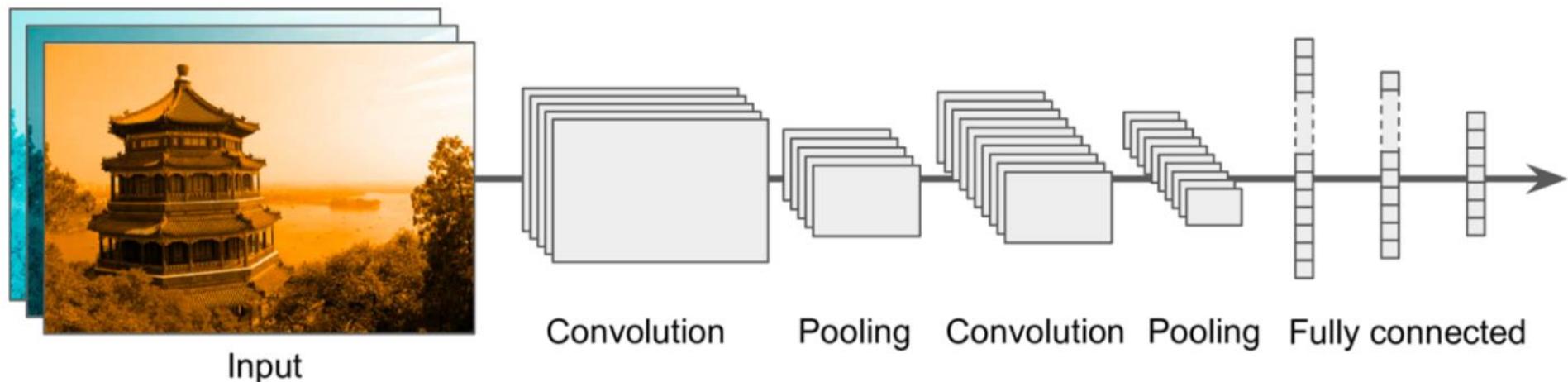


Max Pooling Example



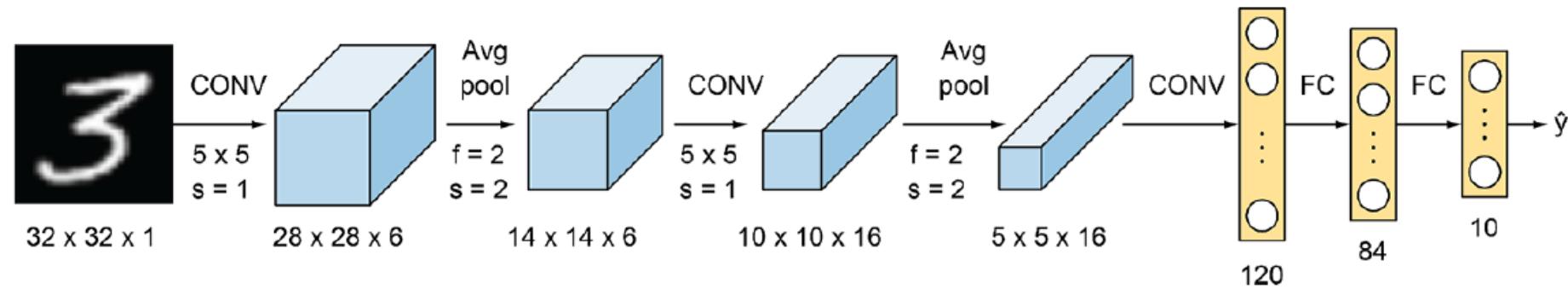
Training CNNs

- Like MLPs, CNNs are trained using SDG and backpropagation
- Large number of parameters need to be determined
- Fairly large training sets are needed (end-to-end vs. given features)
- A regular feedforward neural network is added composed of fully connected layers
- Final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities)

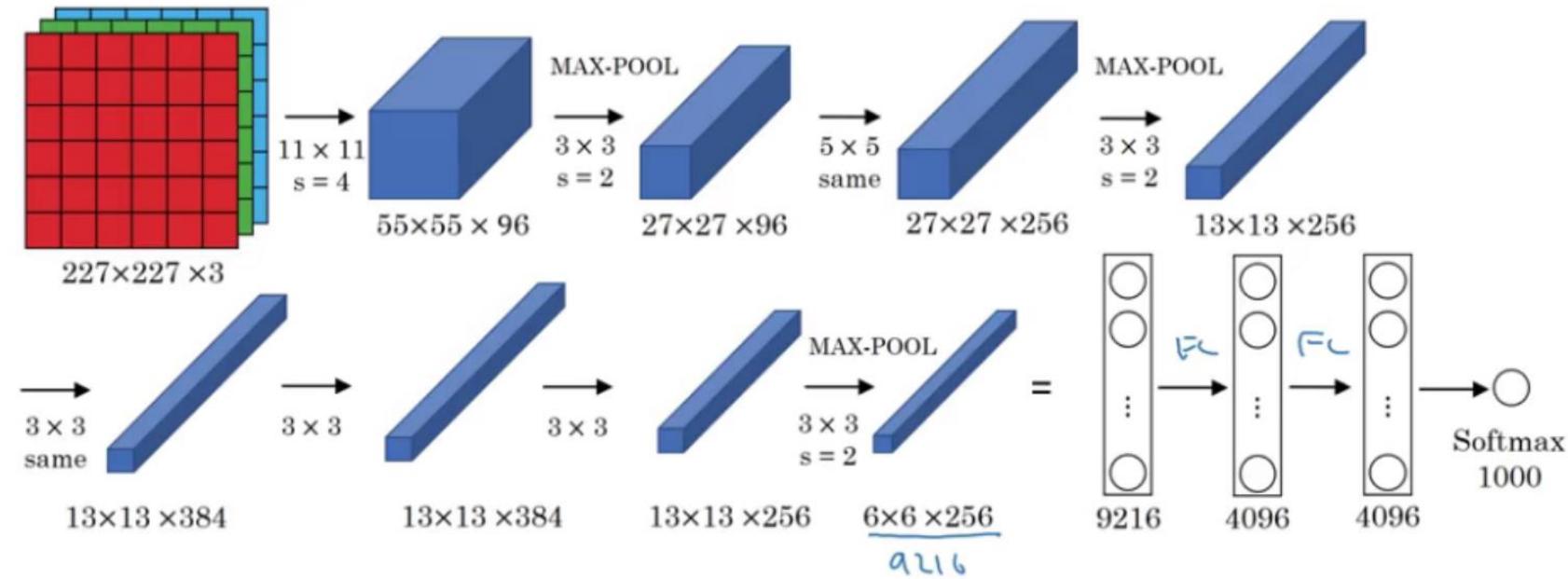


LeNet-5 by LeCun et al., 1989

- First convolutional network proposed by Yann LeCun et al.
- Recognition of handwritten digits
- Outperformed all other networks (at that time)



AlexNet (2012)

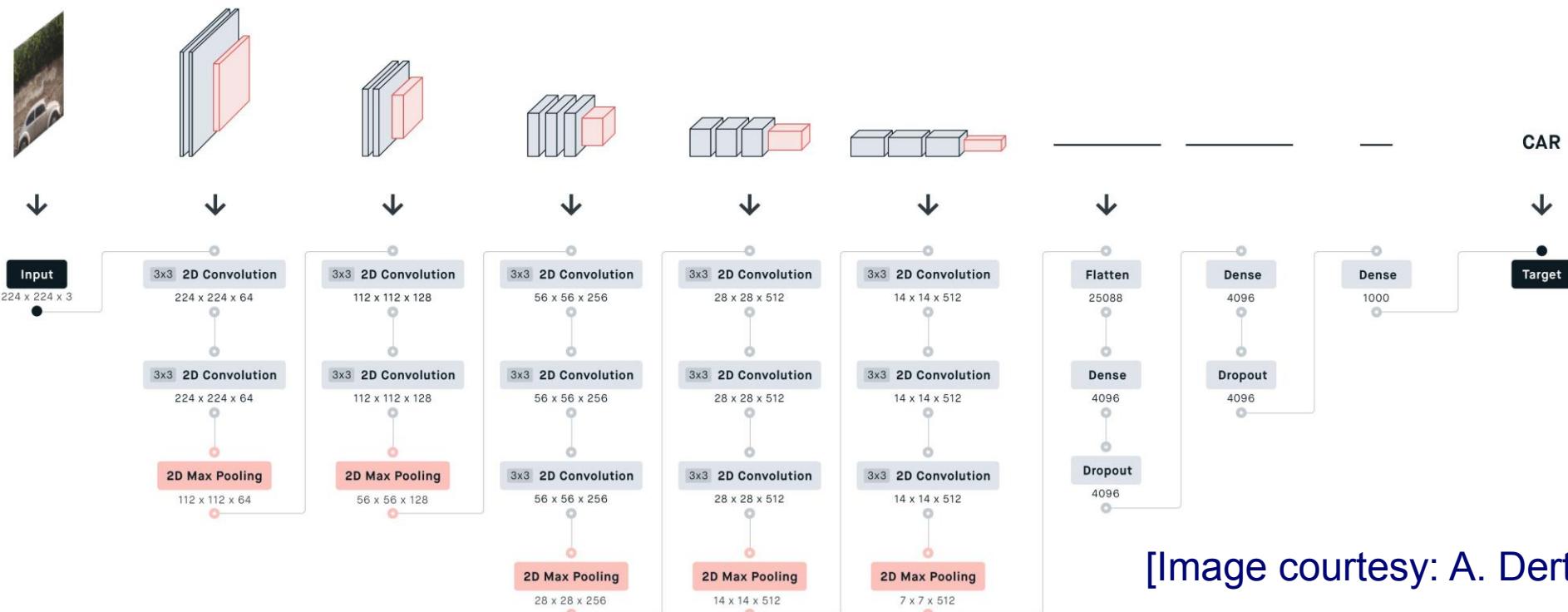


This network is similar to LeNet-5 with just more convolution and pooling layers:

- **Parameters:** 60 million
- **Activation function:** ReLu

VGG-16/19 (2014)

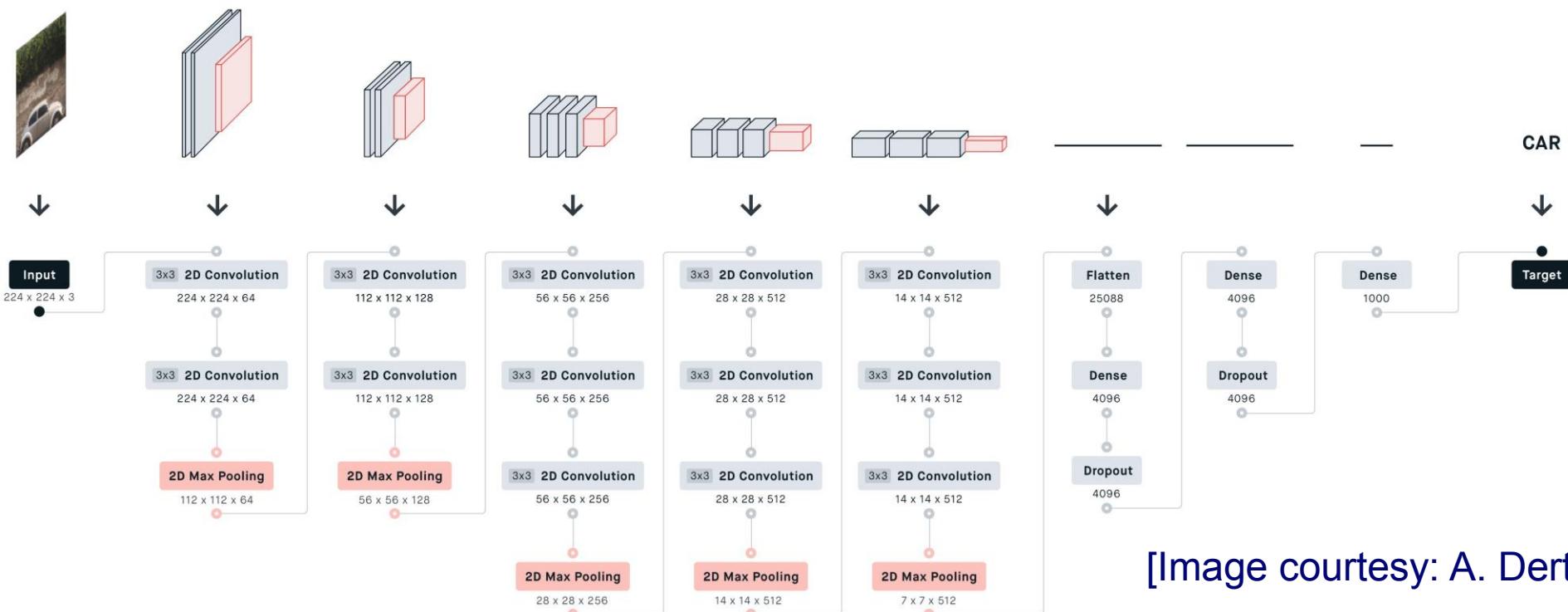
- Stacks elements from AlexNet using smaller filters
- 16/19 layers
- 138M parameters (16 layer version)



[Image courtesy: A. Dertat]

VGG-16/19 (2014)

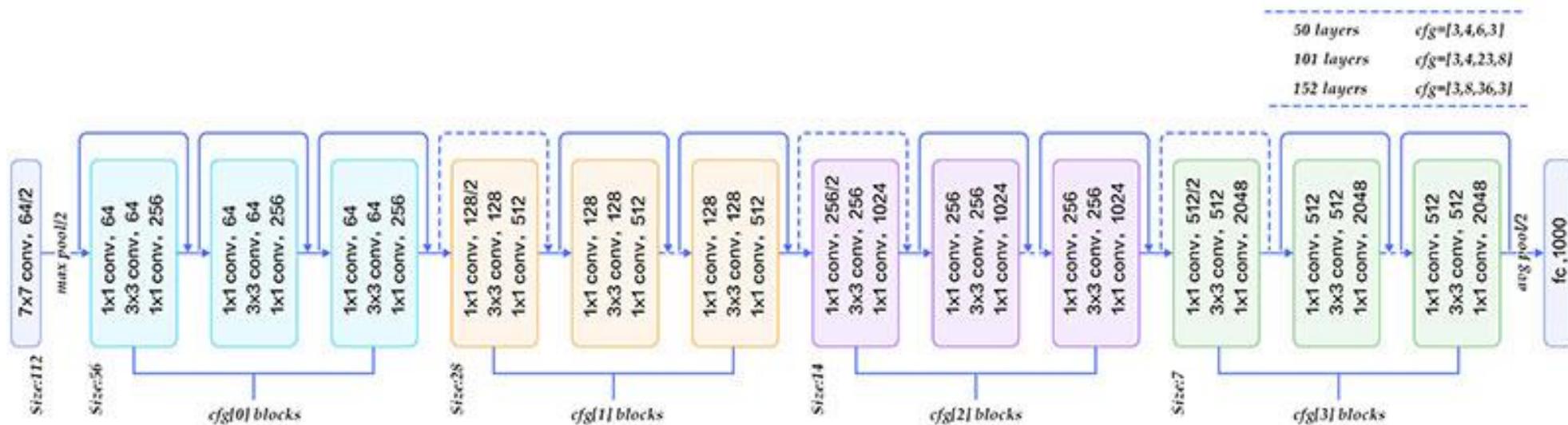
- Stacks elements from AlexNet using smaller filters
- 16/19 layers
- 138M parameters (16 layer version)



[Image courtesy: A. Dertat]

VGG-16/19 (2014)

- Very deep network: 152 layers
- Consists of residual blocks



[Image courtesy: A. Dertat]

Data-intensive space engineering

Lecture 10

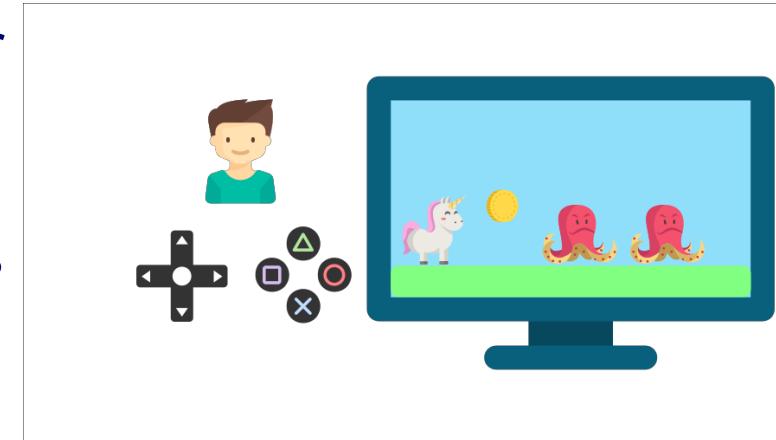
Carlos Sanmiguel Vila

Based on Deep RL Course from
Hugging Face



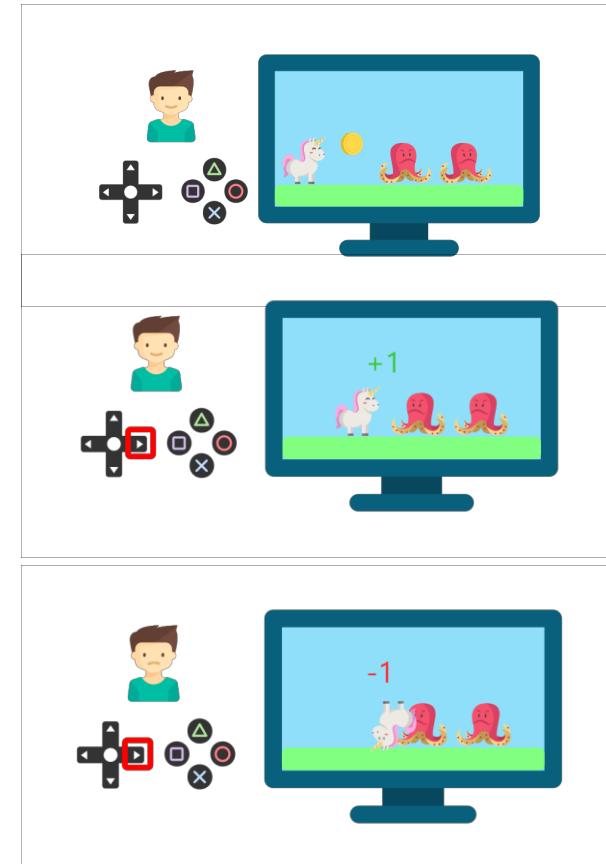
What is Reinforcement Learning?

- The idea behind Reinforcement Learning is that an agent (an AI) will learn from the environment by interacting with it (through trial and error) and receiving rewards (negative or positive) as feedback for performing actions.
- Learning from interactions with the environment comes from our natural experiences.
- For instance, imagine a child in front of a video game he never played. Giving him a controller and leaving him alone.



What is Reinforcement Learning?

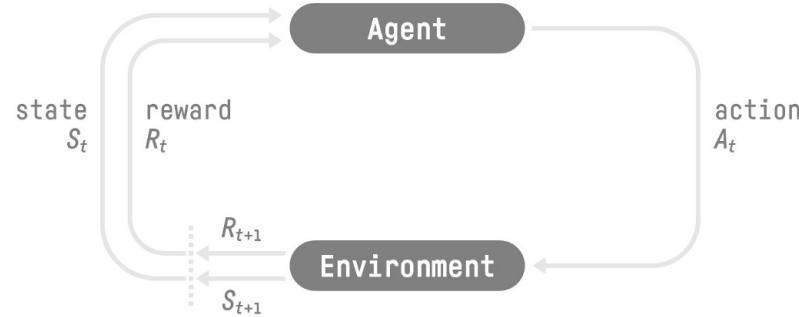
- He will interact with the environment (the video game) by pressing the right button (action). He got a coin, that's a +1 reward. It's positive, he just understood that in this game **he must get the coins**. But then, **he presses the right button again** and he touches an enemy. He just died, so that's a -1 reward.
- By interacting with his environment through trial and error, your little brother understands that he needs to get coins in this environment but avoid the enemies. Without any supervision, the child will get better and better at playing the game.
- That's how humans and animals learn, through interaction. Reinforcement Learning is just a computational approach of learning from actions.



What is Reinforcement Learning?

We can now make a formal definition:

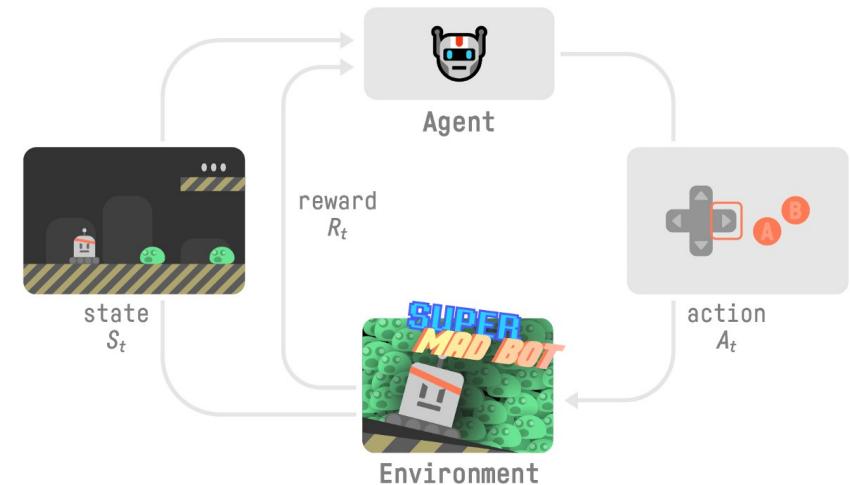
Reinforcement learning is a framework for solving control tasks (also called decision problems) by building agents that learn from the environment by interacting with it through trial and error and receiving rewards (positive or negative) as unique feedback.



The RL Process

To understand the RL process, let's imagine an agent learning to play a platform game:

- Our Agent receives **state S_0** from the **Environment** — we receive the first frame of our game (Environment).
- Based on that **state S_0** , the Agent takes **action A_0** — our Agent will move to the right.
- The environment goes to a **new state S_1** — new frame.
- The environment gives some **reward R_1** to the Agent — we're not dead (*Positive Reward +1*).



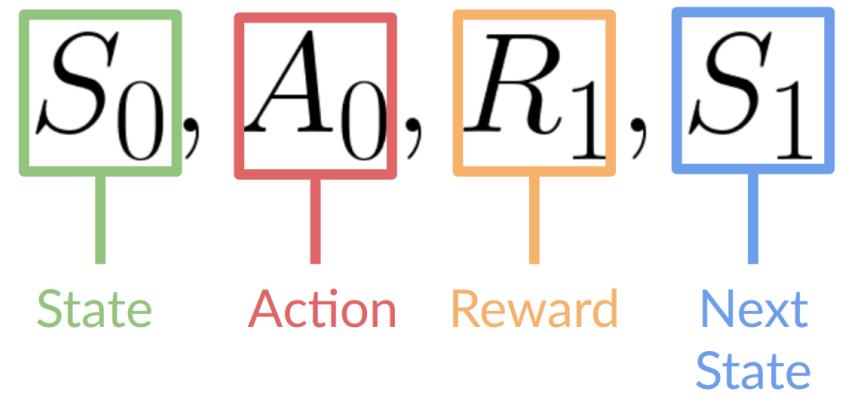
The RL Process

This RL loop outputs a sequence of **state, action, reward and next state**.

The agent's goal is to maximize its cumulative reward, **called the expected return**.

Why is the goal of the agent to maximize the expected return?

- Because RL is based on the reward hypothesis, which is that all goals can be described as the maximization of the expected return (expected cumulative reward).
- That's why in Reinforcement Learning, to have the best behavior, we aim to learn to take actions that maximize the expected cumulative reward.



The RL Process

In papers, you'll see that the RL process is called a **Markov Decision Process** (MDP).

We'll talk again about the Markov Property in the following slides. But if you need to remember something today about it, it's this: the Markov Property implies that our agent needs **only the current state to decide** what action to take and **not the history of all the states and actions** they took before.

Observations/States Space

Observations/States are the **information our agent gets from the environment**. In the case of a video game, it can be a frame (a screenshot). In the case of the trading agent, it can be the value of a certain stock, etc.

There is a differentiation to make between *observation* and *state*, however:

- State s : is a **complete description of the state of the world** (there is no hidden information). In a fully observed environment.
- In a chess game, we have access to the whole board information, so we receive a state from the environment. In other words, the environment is fully observed.



Observations/States Space

Observations/States are the **information our agent gets from the environment**. In the case of a video game, it can be a frame (a screenshot). In the case of the trading agent, it can be the value of a certain stock, etc.

There is a differentiation to make between *observation* and *state*, however:

- Observation o : is a **partial description of the state**. In a partially observed environment.
- In Super Mario Bros, we only see the part of the level close to the player, so we receive an observation. In Super Mario Bros, we are in a partially observed environment. We receive an observation since we only see a part of the level.



Action Space

The Action space is the set of **all possible actions in an environment**.

The actions can come from a *discrete* or *continuous* space:

- *Discrete space*: the number of possible actions is **finite**. Again, in Super Mario Bros, we have a finite set of actions since we have only 4 directions.
- *Continuous space*: the number of possible actions is infinite. A Self-Driving Satellite Mega-Constellation has infinite possible actions since it must perform whatever orbit modification is required.

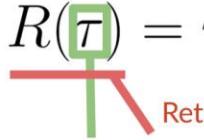


Rewards and the discounting

The reward is fundamental in RL because it's the only feedback for the agent. Thanks to it, our agent knows if the action taken was good or not.

The cumulative reward at each time step t can be written as:

$$R(\tau) = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + \dots$$

 Return: cumulative reward

Trajectory (read Tau)
Sequence of states and actions

Which is equivalent to:

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

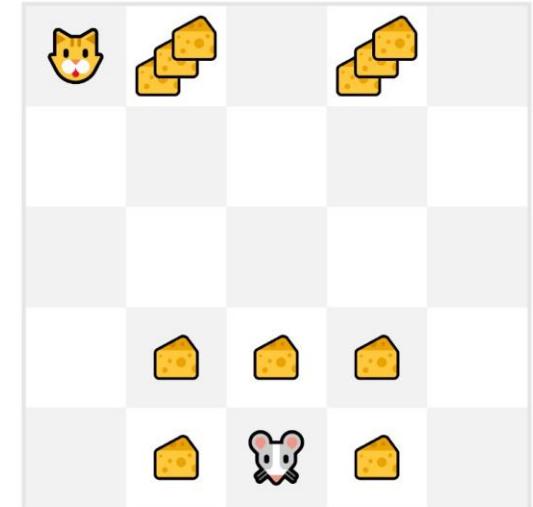
Rewards and the discounting

However, in reality, **we can't just add them like that**. The rewards that come sooner (at the beginning of the game) **are more likely to happen** since they are more predictable than the long-term future reward.

Let's say your agent is this tiny mouse that can move one tile each time step, and your opponent is the cat (that can move too). The mouse's goal is to eat the maximum amount of cheese before being eaten by the cat.

As we can see in the diagram, it's more probable to eat the cheese near us than the cheese close to the cat (the closer we are to the cat, the more dangerous it is).

Consequently, the reward near the cat, even if it is bigger (more cheese), will be more discounted since we're not really sure we'll be able to eat it.



Rewards and the discounting

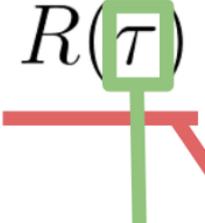
To discount the rewards, we proceed like this:

1. We define a discount rate called gamma. **It must be between 0 and 1.** Most of the time between 0.95 and 0.99.
 - The larger the gamma, the smaller the discount. This means our agent **cares more about the long-term reward.**
 - On the other hand, the smaller the gamma, the bigger the discount. This means our **agent cares more about the short term reward (the nearest cheese).**
2. Then, each reward will be discounted by gamma to the exponent of the time step. As the time step increases, the cat gets closer to us, so the future reward is less and less likely to happen.

Rewards and the discounting

Our discounted expected cumulative reward is:

$$R(\tau) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots$$

 Return: cumulative reward Gamma: discount rate

Trajectory (read Tau)
Sequence of states and actions

$$R(\tau) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Type of tasks

A task is an **instance** of a Reinforcement Learning problem. We can have two types of tasks: **episodic** and **continuing**.

Episodic task

In this case, we have a starting point and an ending point (**a terminal state**). This creates an **episode**: a list of States, Actions, Rewards, and new States. For instance, think about Super Mario Bros: an episode begins at the launch of a new Mario Level and ends **when you're killed or you reached the end of the level**.

Continuing tasks

These are tasks that continue forever (**no terminal state**). In this case, the agent must **learn how to choose the best actions and simultaneously interact with the environment**. For instance, an agent that does orbit maintenance. For this task, there is no starting point and terminal state. **The agent keeps running until we decide to stop it.**



The Exploration/Exploitation trade-off

Finally, before looking at the different methods to solve Reinforcement Learning problems, we must cover one more very important topic: *the exploration/exploitation trade-off*.

- *Exploration* is exploring the environment by trying random actions in order to **find more information about the environment**.
- *Exploitation* is **exploiting known information to maximize the reward**.

Remember, the goal of our RL agent is to maximize the expected cumulative reward. However, **we can fall into a common trap**.

The Exploration/Exploitation trade-off

Let's take an example. In this game, our mouse can have an **infinite amount of small cheese** (+1 each). But at the top of the maze is a gigantic sum of cheese (+1000).

- If we only focus on exploitation, our agent will never reach the gigantic sum of cheese. Instead, it will only exploit **the nearest source of rewards**, even if this source is small (exploitation).
- But if our agent does a little bit of exploration, it can **discover the big reward** (the pile of big cheese).



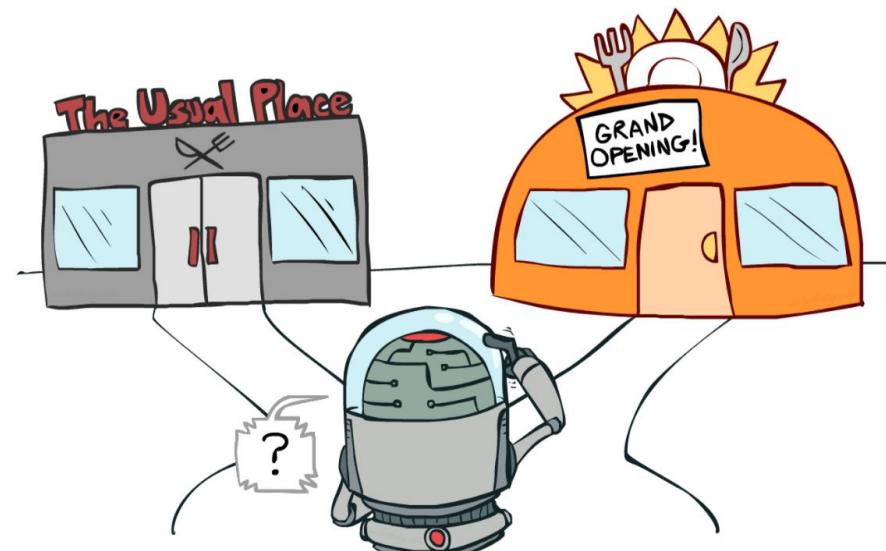
This is what we call the exploration/exploitation trade-off. We need to balance how much we **explore the environment** and how much we **exploit what we know about the environment**. Therefore, we must **define a rule that helps to handle this trade-off**.

The Exploration/Exploitation trade-off

If it's still confusing, think of a real problem: the choice of picking a restaurant:

Exploitation: You go to the same one that you know is good every day **and take the risk to miss another better restaurant.**

Exploration: Try restaurants you never went to before, with the risk of having a bad experience **but the probable opportunity of a fantastic experience.**

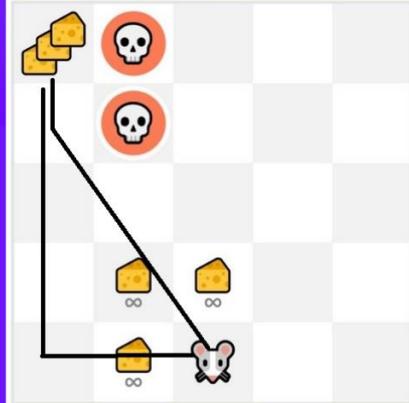


C.Sanmiguel Vila

The Exploration/Exploitation trade-off

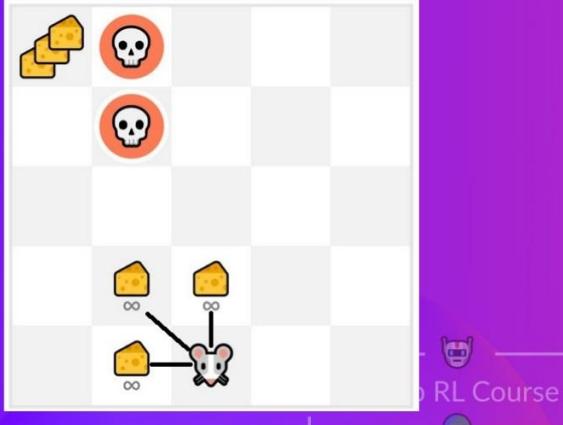
Exploration/ Exploitation tradeoff

Exploration: trying random actions in order to find more information about the environment.



A 4x4 grid-based environment diagram. In the top-left corner, there is a stack of three cheese icons. To its right are two red circles containing white skull icons. In the bottom-right corner, there is a small mouse icon facing right. Two arrows originate from the bottom-right corner: one points up to the top-right red circle, and another points left to the bottom-right red circle. Each of these two red circles has a value of "∞" written below it. The grid consists of light gray squares.

Exploitation: using known information to maximize the reward.



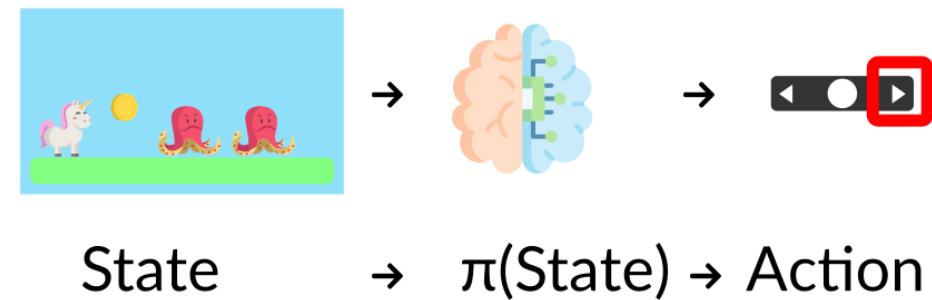
A 4x4 grid-based environment diagram. In the top-left corner, there is a stack of three cheese icons. To its right are two red circles containing white skull icons. In the bottom-right corner, there is a small mouse icon facing right. Two arrows originate from the bottom-right corner: one points up to the top-right red circle, and another points left to the bottom-right red circle. Each of these two red circles has a value of "∞" written below it. The grid consists of light gray squares. At the bottom right, there is a circular button labeled "Go to RL Course".

Two main approaches for solving RL problems

How do we build an RL agent that can **select the actions that maximize its expected cumulative reward?**

The Policy π : the agent's brain

The Policy π is the **brain of our Agent**, it's the function that tells us what **action to take given the state we are in**. So it **defines the agent's behavior** at a given time.



Two main approaches for solving RL problems

This Policy **is the function we want to learn**, our goal is to find the optimal policy π^* , the policy that **maximizes expected return** when the agent acts according to it. We find this π^* **through training**.

There are two approaches to train our agent to find this optimal policy π^* :

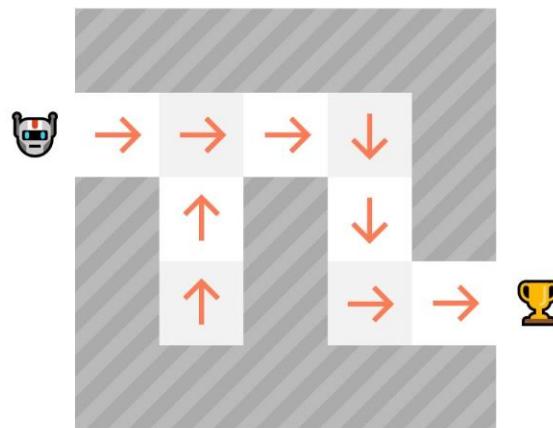
- **Directly**, by teaching the agent to learn which **action to take**, given the current state: **Policy-Based Methods**.
- Indirectly, **teach the agent to learn which state is more valuable** and then take the action that **leads to the more valuable states**: Value-Based Methods.

Policy-Based Methods

In Policy-Based methods, **we learn a policy function directly.**

This function will define a mapping from each state to the best corresponding action. Alternatively, it could define **a probability distribution over the set of possible actions at that state.**

As we can see here, the policy (deterministic) **directly indicates the action to take for each step.**

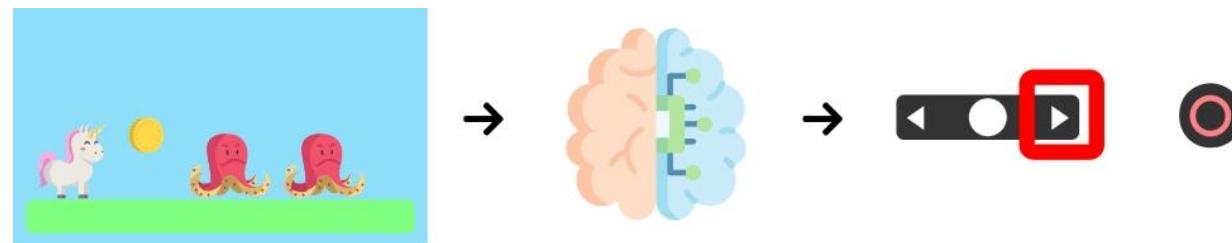


Policy-Based Methods

We have two types of policies:

- *Deterministic*: a policy in a given state **will always return the same action**.

$$a = \pi(s)$$



State s_0 → $\pi(s_0)$ → $a_0 = \text{Right}$

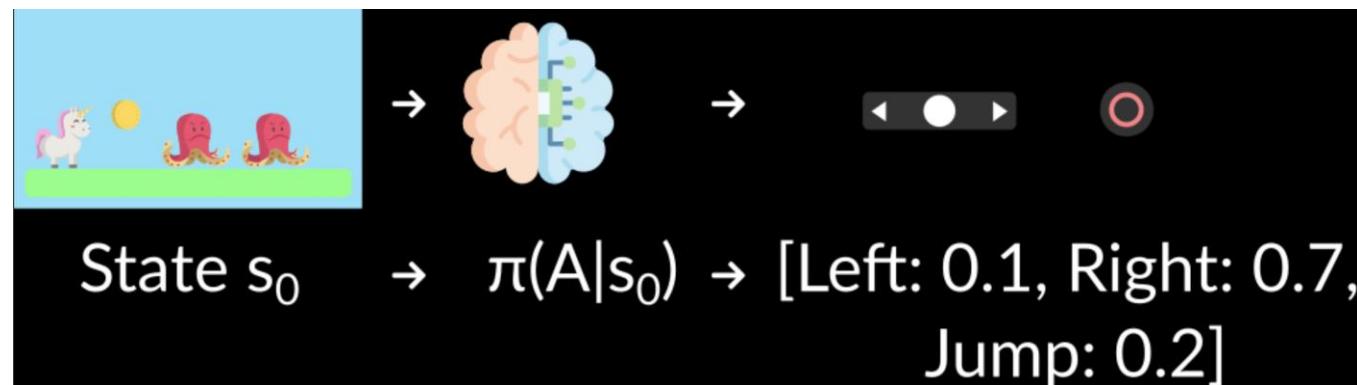
Policy-Based Methods

We have two types of policies:

- *Stochastic*: outputs a **probability distribution over actions**.

$$\pi(a|s) = P[A|s]$$

Probability Distribution over the set of actions given the state



Value-based methods

In value-based methods, instead of learning a policy function, we **learn a value function** that maps a state to the expected value **of being at that state**.

The value of a state is the **expected discounted return** the agent can get if it **starts in that state, and then acts according to our policy**.

“Act according to our policy” just means that our policy is “**going to the state with the highest value**”.

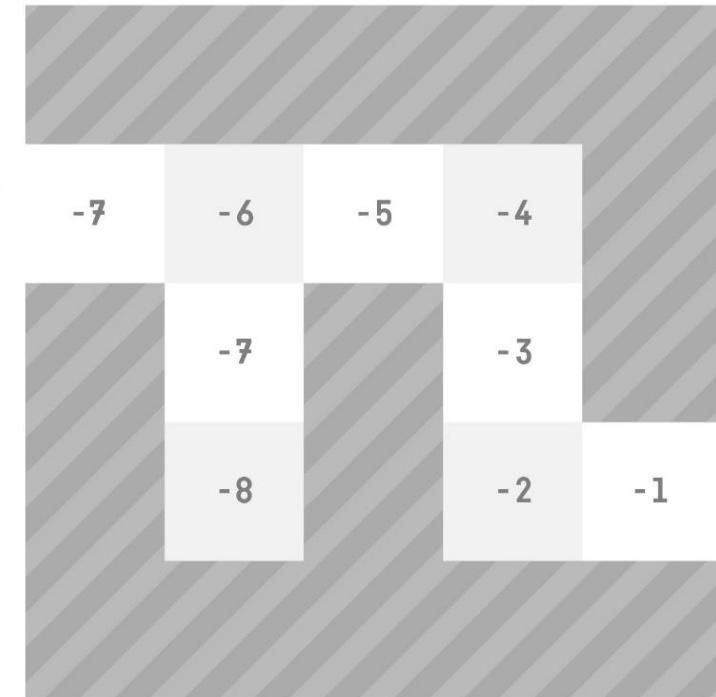
$$\underline{v_\pi(s)} = \mathbb{E}_\pi \left[\underline{R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots} \mid \underline{S_t = s} \right]$$

Value function Expected discounted return Starting at state s

Value-based methods

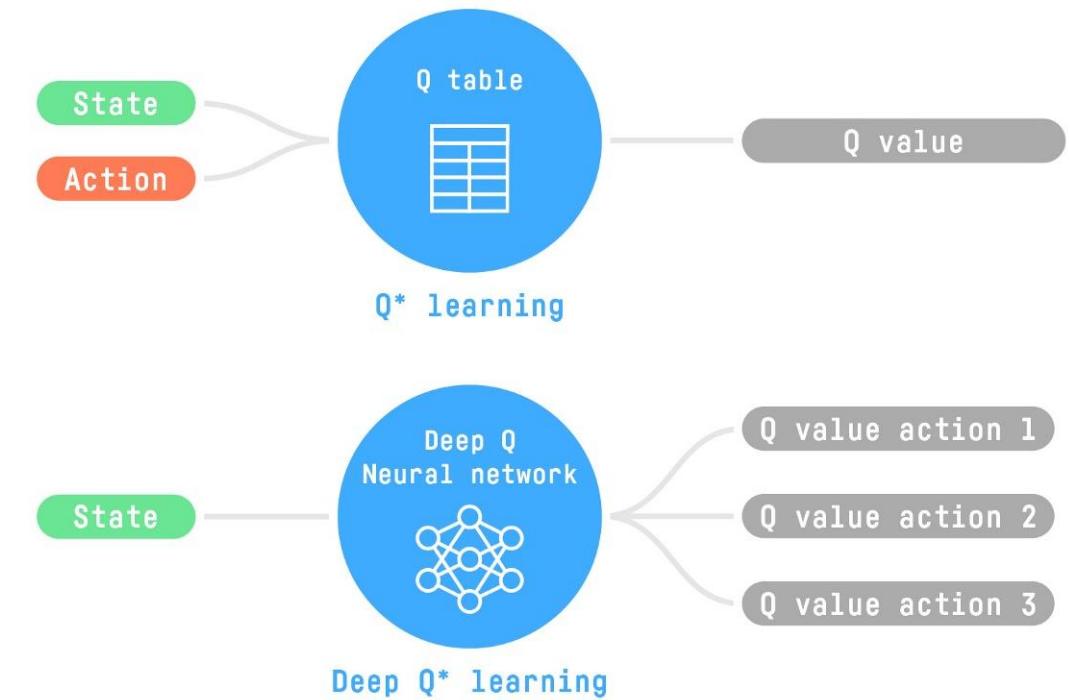
Here we see that our value function **defined values for each possible state.**

Thanks to our value function, at each step our policy will select the state with the biggest value defined by the value function: -7, then -6, then -5 (and so on) to attain the goal.



The “Deep” in Reinforcement Learning

- Deep Reinforcement Learning introduces deep neural networks to solve Reinforcement Learning problems — hence the name “deep”. For instance, we’ll learn about two value-based algorithms: Q-learning (classic Reinforcement Learning) and then Deep Q-learning.
- In the first approach, we use a traditional algorithm to create a Q table that helps us find what action to take for each state. In the second approach, we will use a Neural Network (to approximate the Q value).



Summary

RL is a computational approach of learning from actions. We build an agent that learns from the environment **by interacting with it through trial and error** and receiving rewards (negative or positive) as feedback.

The goal of any RL agent is to maximize its expected cumulative reward (also called expected return) because RL is based on the **reward hypothesis**, which is that **all goals can be described as the maximization of the expected cumulative reward**.

The RL process is a loop that outputs a sequence of **state, action, reward and next state**. To calculate the expected cumulative reward (expected return), we discount the rewards: the rewards that come sooner (at the beginning of the game) are **more probable to happen since they are more predictable than the long term future reward**.

Summary

To solve an RL problem, you want to **find an optimal policy**. The policy is the “brain” of your agent, which will tell us **what action to take given a state**. The optimal policy is the one which **gives you the actions that maximize the expected return**.

There are two ways to find your optimal policy:

- By training your policy directly: **policy-based methods**.
- By training a value function that tells us the expected return the agent will get at each state and use this function to define our policy: **value-based methods**.

Finally, we speak about Deep RL because we introduce **deep neural networks to estimate the action to take (policy-based) or to estimate the value of a state (value-based)** hence the name “deep”.

Data-intensive space engineering

Lecture 11

Carlos Sanmiguel Vila

Based on Deep RL Course from
Hugging Face



How to find optimal policy π^* ?

- In value-based methods, **we learn a value function that maps a state to the expected value of being at that state.**
- The value of a state is the **expected discounted return** the agent can get if it **starts at that state and then acts according to our policy.**

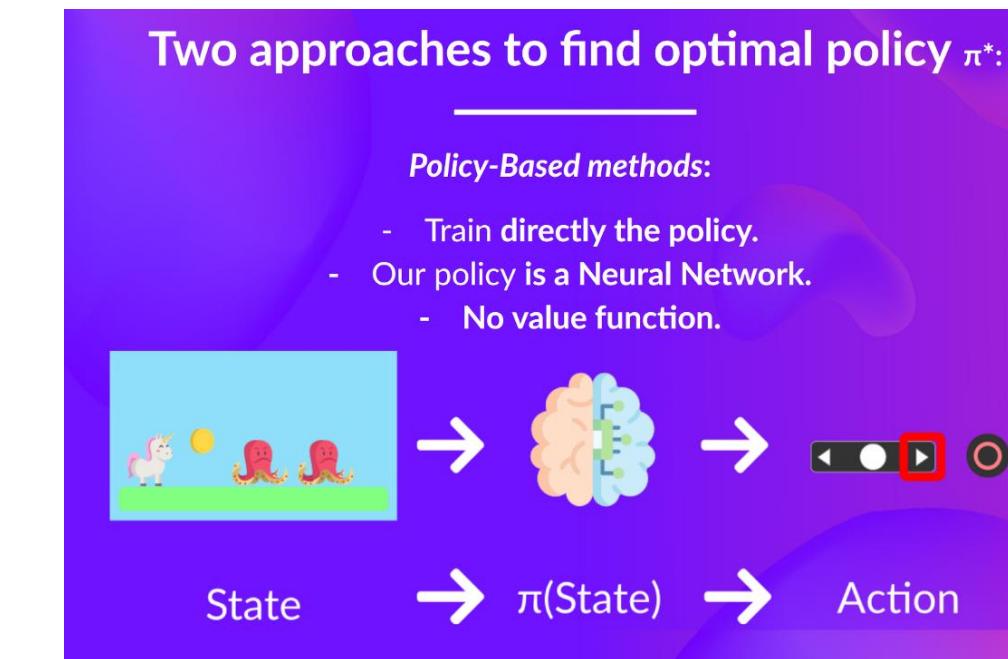
$$\underline{v_\pi(s)} = \underline{\mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s]}$$

Value
function Expected discounted return Starting
at state s

But what does it mean to act according to our policy? After all, we don't have a policy in value-based methods since we train a value function and not a policy.

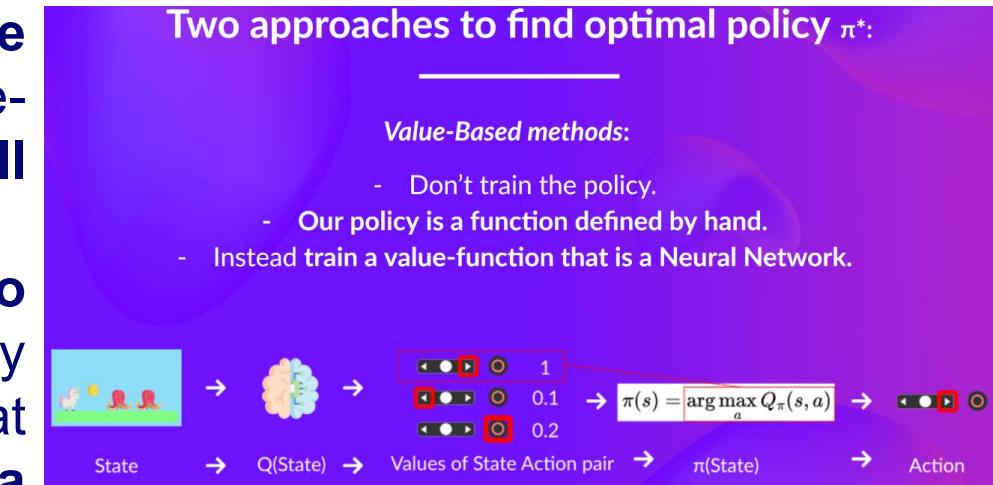
How to find optimal policy π^* ?

- To find the optimal policy, we learned about **two different methods**:
- *Policy-based methods:* **Directly train the policy** to select what action to take given a state (or a probability distribution over actions at that state). In this case, we **don't have a value function**.
- The policy takes a state as input and outputs what action to take at that state (deterministic policy: a policy that output one action given a state, contrary to stochastic policy that output a probability distribution over actions).
- And consequently, **we don't define by hand the behavior of our policy; it's the training that will define it.**



How to find optimal policy π^* ?

- To find the optimal policy, we learned about **two different methods**:
- *Value-based methods*: **Indirectly, by training a value function** that outputs the value of a state or a state-action pair. Given this value function, our policy **will take an action**.
- Since the policy is not trained/learned, **we need to specify its behavior**. For instance, if we want a policy that, given the value function, will take actions that always lead to the biggest reward, **we'll create a Greedy Policy**.
- Consequently, whatever method you use to solve your problem, **you will have a policy**. In the case of value-based methods, you don't train the policy: your policy **is just a simple pre-specified function** (for instance, the Greedy Policy) that uses the values given by the value-function to select its actions.



How to find optimal policy π^* ?

So the difference is:

- In policy-based training, **the optimal policy (denoted π^*) is found by training the policy directly.**
- In value-based training, **finding an optimal value function (denoted Q^* or V^*) leads to having an optimal policy.**

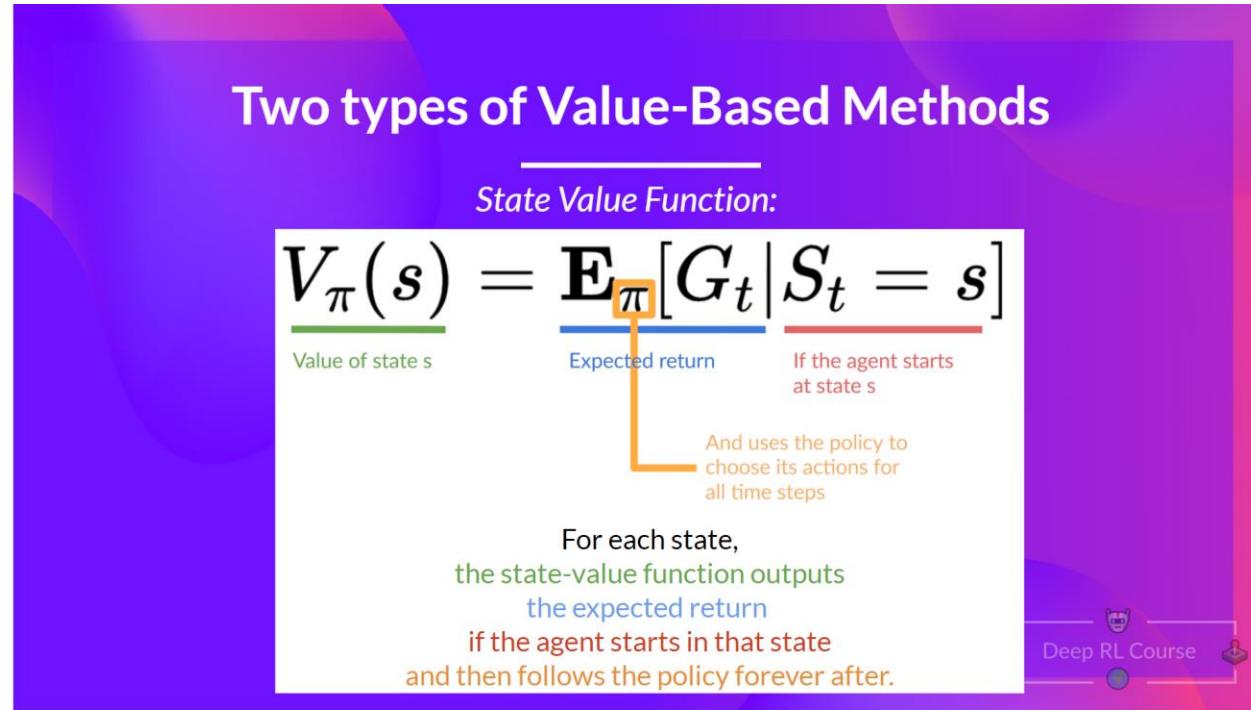
The link between Value and Policy:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

Finding an optimal value function leads to having an optimal policy.

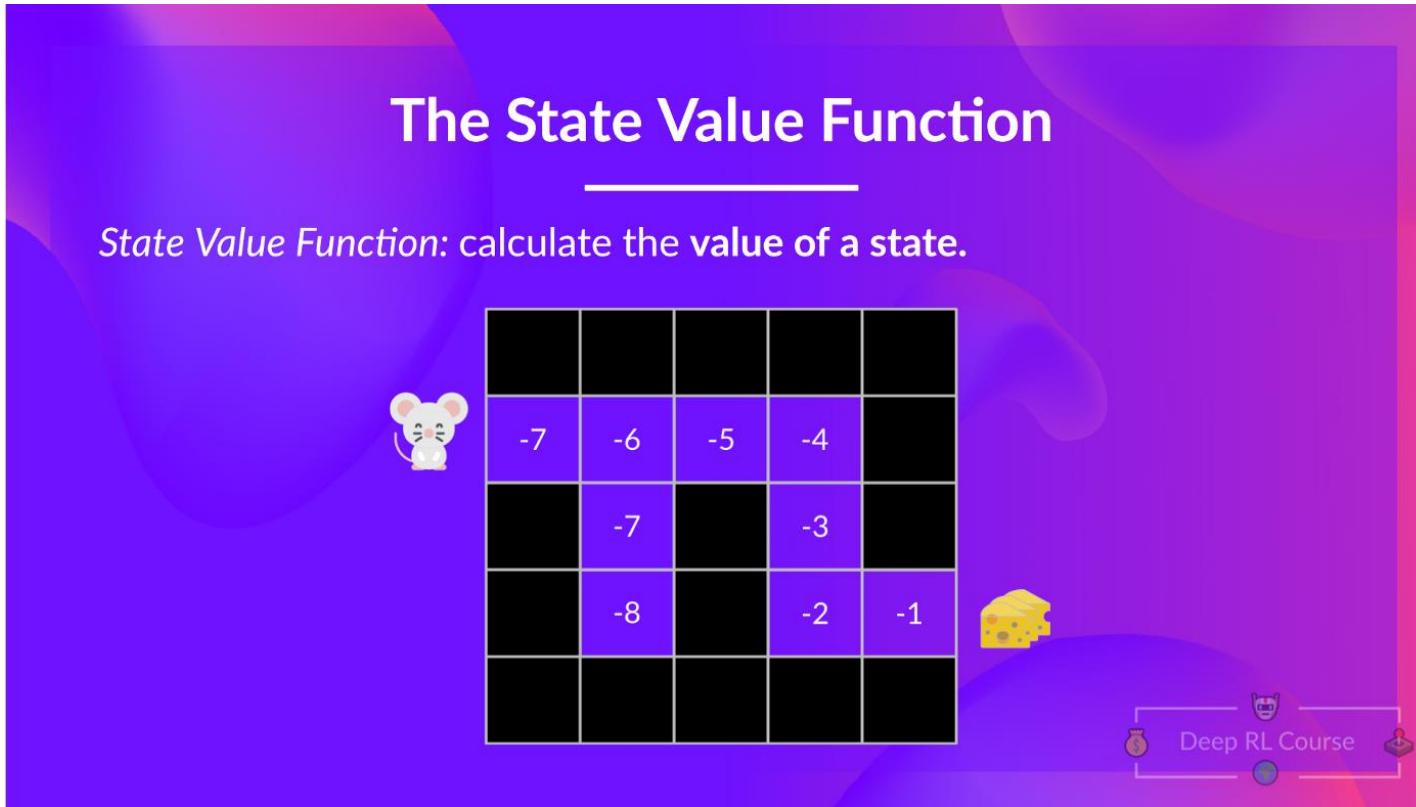
The state-value function

We write the state value function under a policy π like this:



For each state, the state-value function outputs the expected return if the agent **starts at that state** and then follows the policy forever afterward (for all future timesteps).

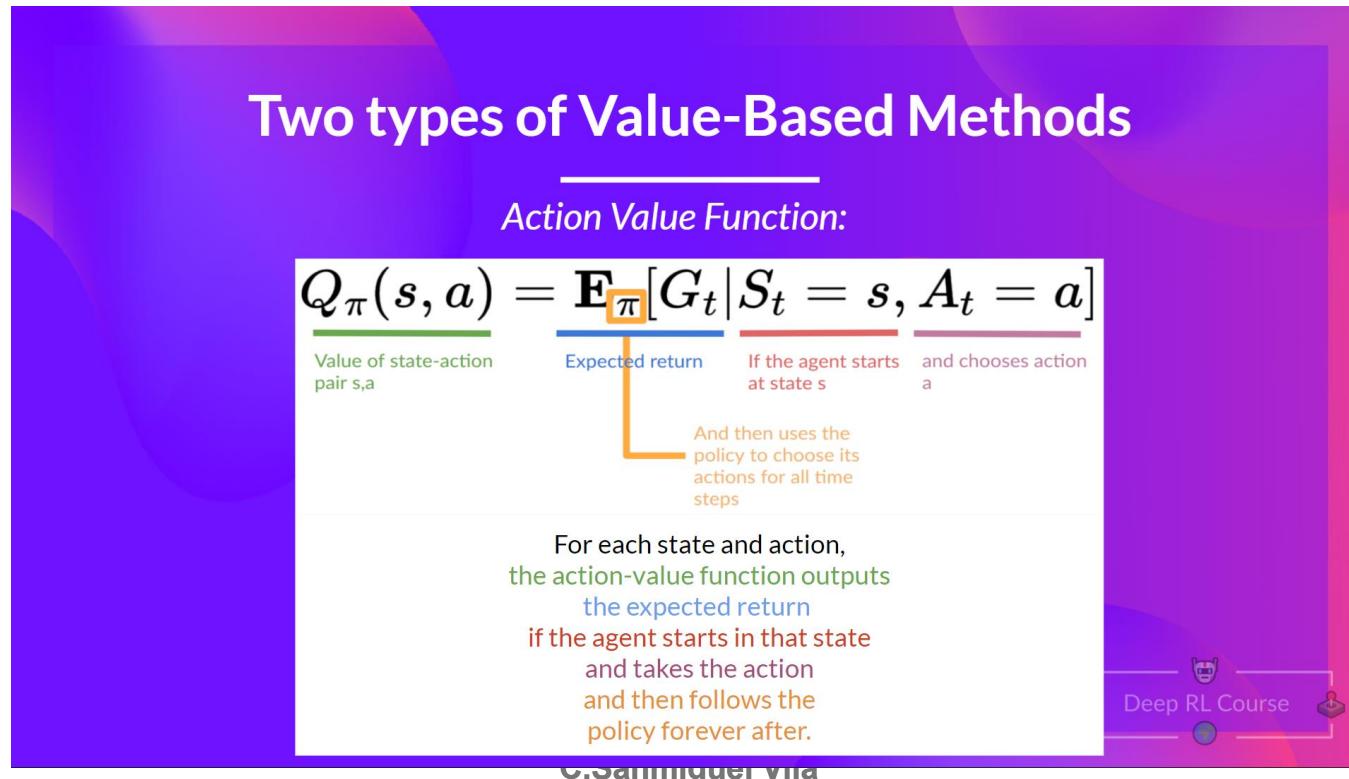
The state-value function



If we take the state with value -7: it's the expected return starting at that state and taking actions according to our policy (greedy policy), so right, right, right, down, down, right, right.

The action-value function

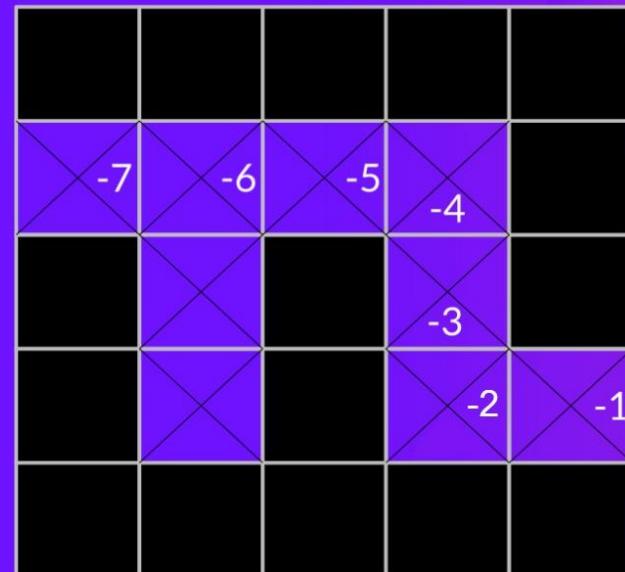
- In the action-value function, for each state and action pair, the action-value function **outputs the expected return** if the agent starts in that state, takes that action, and then follows the policy forever after.
- The value of taking action a in state s under a policy π is:



The action-value function

The Action Value Function

Action Value Function: calculate the value of state-action pair.

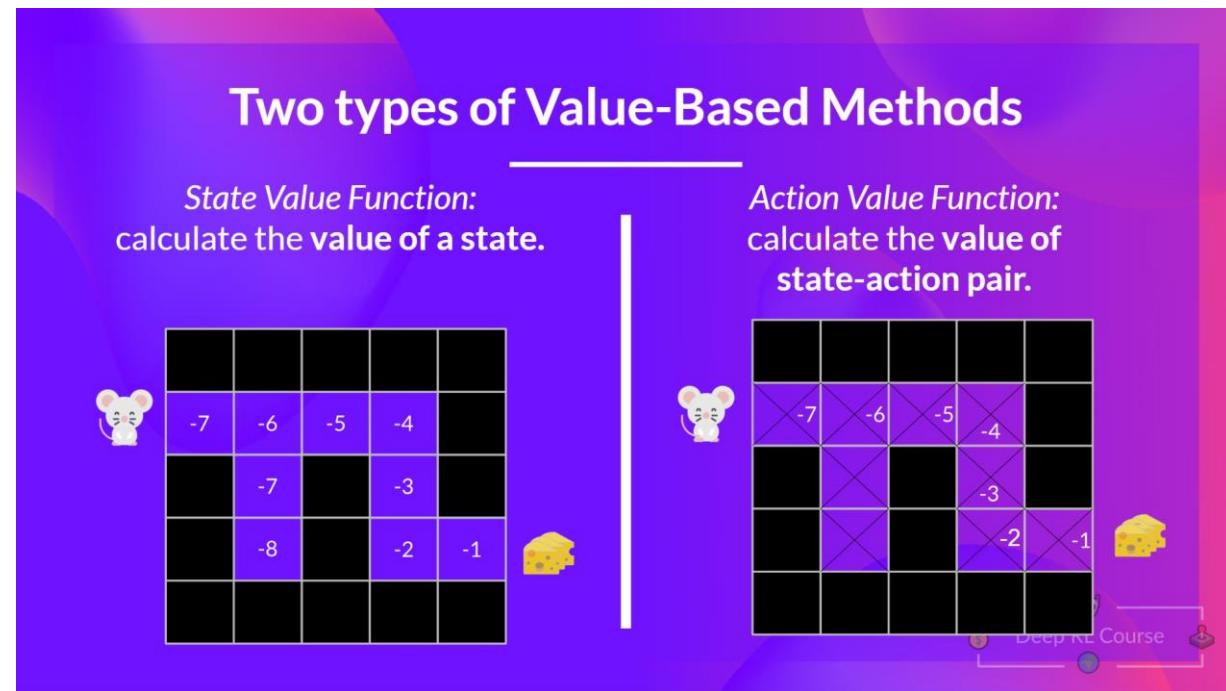


*We didn't fill all the state-actions pair for the example of Action-value function



Value-based methods

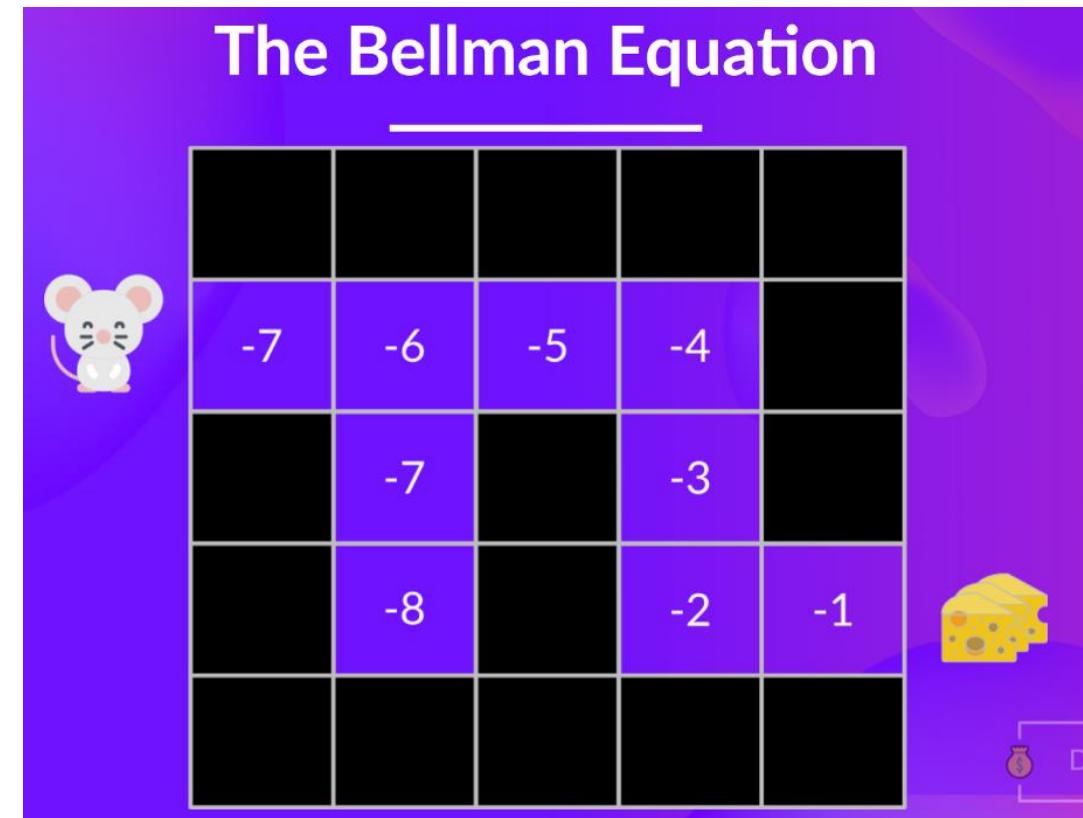
- For the state-value function, we calculate **the value of a state S_t**
- For the action-value function, we calculate **the value of the state-action pair (S_t, A_t) hence the value of taking that action at that state.**
- In either case, whichever value function we choose (state-value or action-value function), **the returned value is the expected return.**
- However, the problem is that **to calculate EACH value of a state or a state-action pair, we need to sum all the rewards an agent can get if it starts at that state.**



The Bellman Equation

The Bellman equation **simplifies our state value or state-action value calculation.**

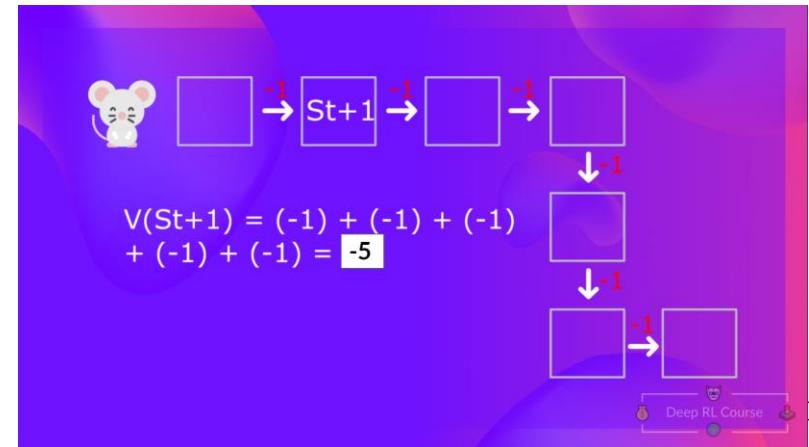
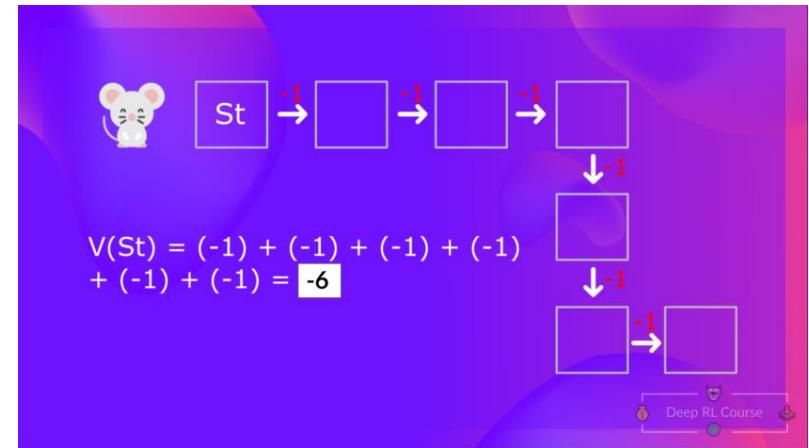
- With what we have learned so far, we know that if we calculate $V(S_t)$ (the value of a state), we need to calculate the return starting at that state and then follow the policy forever after. (The policy we defined in the following example is a Greedy Policy; for simplification, we don't discount the reward).
- A greedy algorithm reaches a problem solution using sequential steps where, at each step, it makes a decision based on the best solution at that time, without considering future consequences or implications.



The Bellman Equation

To calculate $V(S_t)$ we need to calculate the sum of the expected rewards. Hence:

- To calculate the value of State 1: the sum of rewards if the agent started in that state and then followed the greedy policy (taking actions that lead to the best state values) for all the time steps.
- To calculate the value of State 2: the sum of rewards if the agent started in that state, and then followed the policy for all the time steps..
- We're repeating the computation of the value of different states, which can be tedious if you need to do it for each state value or state-action value.
- Instead of calculating the expected return for each state or each state-action pair, we can use the **Bellman equation**.



The Bellman Equation

The Bellman equation is a recursive equation that works like this: instead of starting from each state from the beginning and calculating the return, we can consider the value of any state as:

The immediate reward R_{t+1} + the discounted value of the state that follows ($\gamma * V(S_{t+1})$)

The Bellman Equation

$$V_\pi(s) = \mathbf{E}_\pi[R_{t+1} + \gamma * V_\pi(S_{t+1}) | S_t = s]$$

Value of state s Expected value of immediate reward + the discounted value of next_state If the agent starts at state s

And uses the policy to choose its actions for all time steps

$$V(St) = R^{t+1} + \text{gamma} * V(St+1)$$

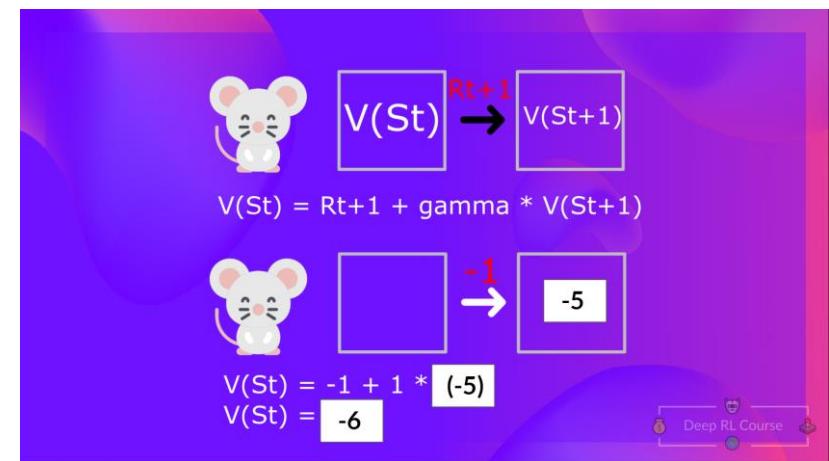
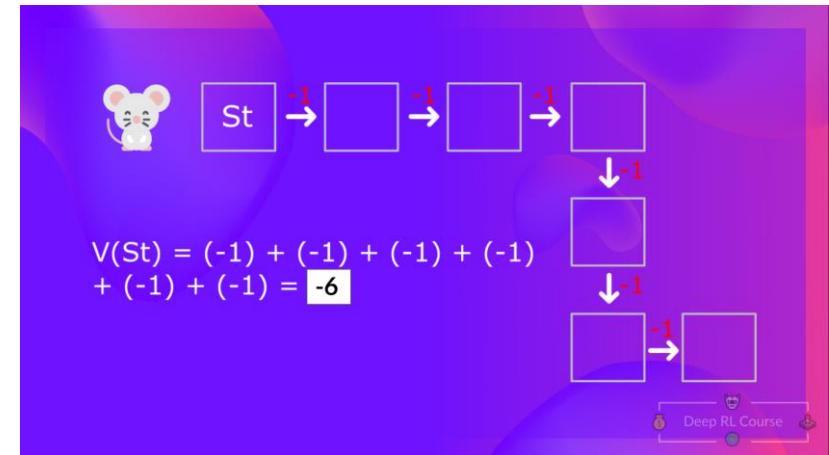
Deep RL Course

The Bellman Equation

If we go back to our example, we can say that the value of State 1 is equal to the expected cumulative return if we start at that state.

- To calculate the value of State 1: the sum of rewards if the agent started in that state 1 and then followed the policy for all the time steps.
- This is equivalent to $V(S_t) = \text{Immediate reward } R_{t+1} + \text{Discounted value of the next state } \gamma * V(S_{t+1})$ (In the interest of simplicity, here we don't discount, so $\gamma = 1$)
- The value of $V(S_{t+1}) = \text{Immediate reward } R_{t+2} + \text{Discounted value of the next state } \gamma * V(S_{t+2})$

To recap, the idea of the Bellman equation is that instead of calculating each value as the sum of the expected return, which is a long process, we calculate the value as the sum of immediate reward + the discounted value of the state that follows.



Monte Carlo vs Temporal Difference Learning

Remember that an RL agent **learns by interacting with its environment**. The idea is that **given the experience and the received reward, the agent will update its value function or policy**.

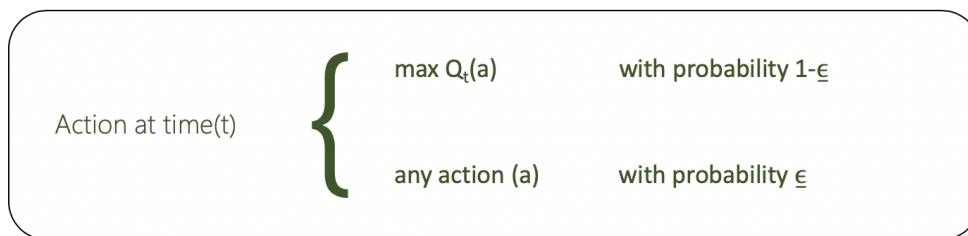
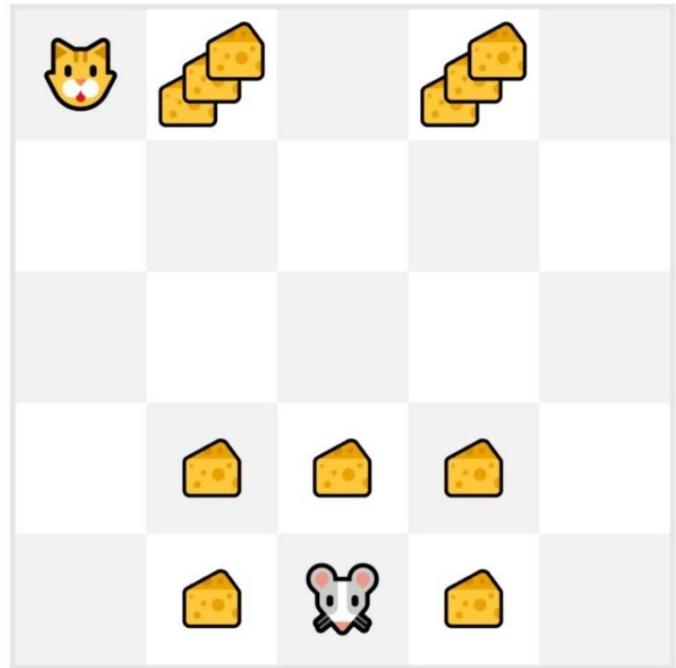
Monte Carlo and Temporal Difference Learning are two different **strategies on how to train our value function or our policy function**. Both **use experience to solve the RL problem**.

On one hand, Monte Carlo uses **an entire episode of experience before learning**. On the other hand, Temporal Difference uses **only a step $(S_t, A_t, R_{t+1}, S_{t+1})$ to learn**.

Monte Carlo

- We always start the episode at the same starting point.
- The agent takes actions using the policy. For instance, using an Epsilon Greedy Strategy, a policy that alternates between exploration (random actions) and exploitation.
- We get the reward and the next state.
- We terminate the episode if the cat eats the mouse or if the mouse moves > 10 steps.
- At the end of the episode, we have a list of State, Actions, Rewards, and Next States tuples
- The agent will sum the total rewards G_t (to see how well it did).
- It will then update $V(S_t)$ based on the formula $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$
- Then start a new game with this new knowledge

By running more and more episodes, the agent will learn to play better and better.



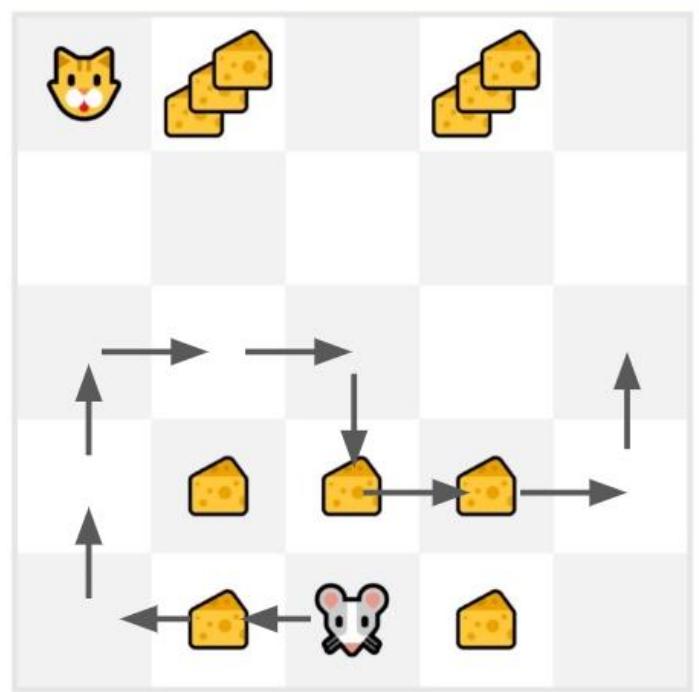
Monte Carlo

For instance, if we train a state-value function using Monte Carlo:

- We initialize our value function **so that it returns 0 value for each state**
- Our learning rate (α) is 0.1 and our discount rate is 1 (= no discount)
- Our mouse **explores the environment and takes random actions**
- The mouse made more than 10 steps, so the episode ends .
- We have a list of state, action, rewards, next_state, **we need to calculate the return**
- $G_t = 0$ $G_t = R_{t+1} + R_{t+2} + R_{t+3} \dots$ (for simplicity, we don't discount the rewards) $G_0 = R_1 + R_2 + R_3 \dots G_0 = R_1 + R_2 + R_3 \dots G_0 = 3$
- We can now compute the **new $V(S_0)$** :

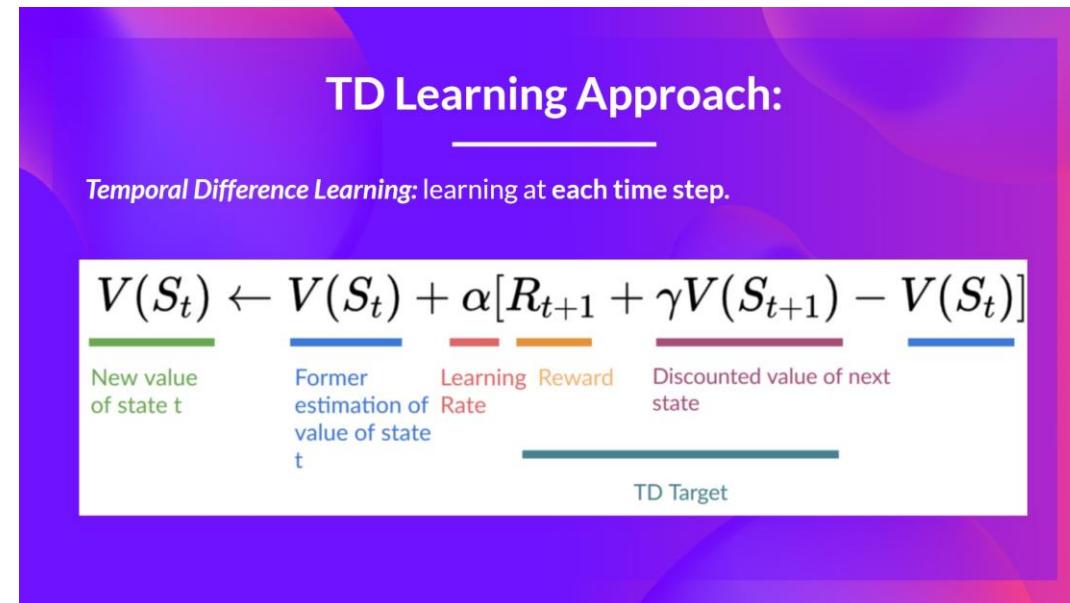
$$V(S_0) = V(S_0) + \alpha[G_0 - V(S_0)]$$

$$V(S_0) = 0 + 0.1[3 - 0] = 0.3$$



Temporal Difference Learning

- Temporal Difference, on the other hand, waits for **only one interaction (one step)** S_{t+1} to form a TD target and update $V(S_t)$ using R_{t+1} and $\gamma * V(S_{t+1})$.
- The idea with TD is to **update the $V(S_t)$ at each step**.
- But because we didn't experience an entire episode, we don't have G_t (expected return). Instead, **we estimate G_t by adding R_{t+1} and the discounted value of the next state**.
- This is called bootstrapping. It's called this **because TD bases its update in part on an existing estimate $V(S_{t+1})$. and not a complete sample G_t** .
- This method is called TD(0) or **one-step TD (update the value function after any individual step)**.



Temporal Difference Learning

If we take the same example

- We initialize our value function so that it returns 0 value for each state.
- Our learning rate (α) is 0.1, and our discount rate is 1 (no discount).
- Our mouse begins to explore the environment and takes a random action: **going to the left**
- It gets a reward $R_{t+1} = 1$ since **it eats a piece of cheese**

We can now update $V(S_0)$:

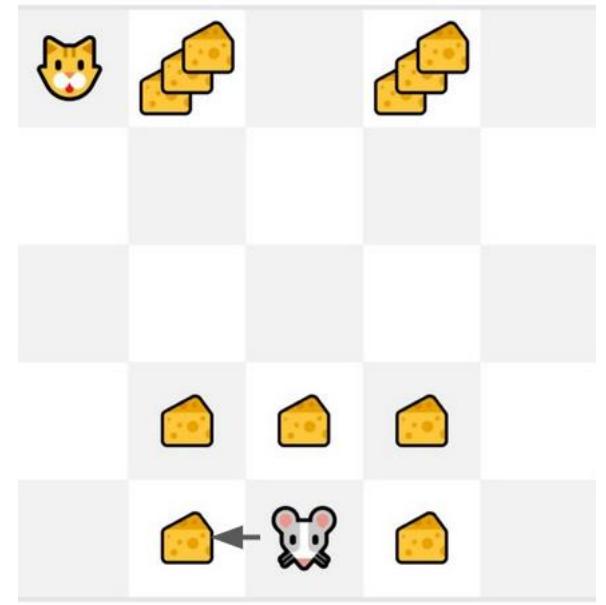
$$\text{New } V(S_0) = V(S_0) + \alpha[R_{t+1} + \gamma * V(S_1) - V(S_0)]$$

$$\text{New } V(S_0) = 0 + 0.1 * [1 + 1 * 0 - 0]$$

$$\text{New } V(S_0) = 0.1.$$

So we just updated our value function for State 0.

Now we **continue to interact with this environment with our updated value function.**



Monte Carlo vs Temporal Difference Learning

To summarize:

With Monte Carlo, we update the value function from a complete episode, and so we **use the actual accurate discounted return of this episode.**

With TD Learning, we update the value function from a step, and we replace G_t , which we don't know, with **an estimated return called the TD target.**

$$\text{Monte Carlo: } V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

$$\text{TD Learning: } V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

What is Q-Learning?

Q-Learning is an **off-policy value-based method that uses a TD approach to train its action-value function**:

- *Off-policy*: we'll talk about that at the end of this unit.
- *Value-based method*: finds the optimal policy indirectly by training a value or action-value function that will tell us **the value of each state or each state-action pair**.
- *TD approach*: **updates its action-value function at each step instead of at the end of the episode**.

Q-Learning is the algorithm we use to train our Q-function, an **action-value function** that determines the value of being at a particular state and taking a specific action at that state.

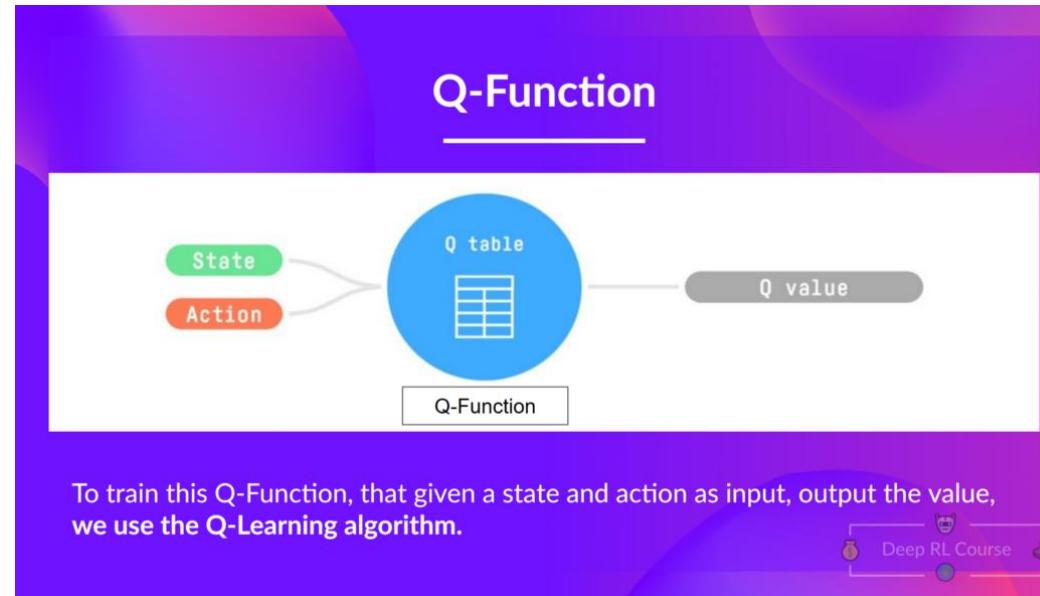
What is Q-Learning?

The Q comes from “the Quality” (the value) of that action at that state.

Let's recap the difference between value and reward:

- The *value* of a state, or a *state-action pair*, is the expected cumulative reward our agent gets if it starts at this state (or state-action pair) and then acts according to its policy.
- The *reward* is the **feedback I get from the environment** after performing an action at a state.

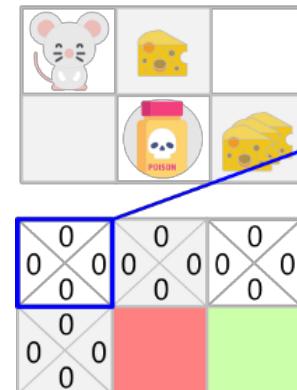
Internally, our Q-function is encoded by a **Q-table**, a **table where each cell corresponds to a state-action pair value**. Think of this Q-table as **the memory or cheat sheet of our Q-function**.



What is Q-Learning?

Let's go through an example of a maze.

The Q-table is initialized. That's why all values are = 0. This table **contains, for each state and action, the corresponding state-action values**. For this simple example, the state is only defined by the position of the mouse. Therefore, we have 2×3 rows in our Q-table, one row for each possible position of the mouse. In more complex scenarios, the state could contain more information than the position of the actor.



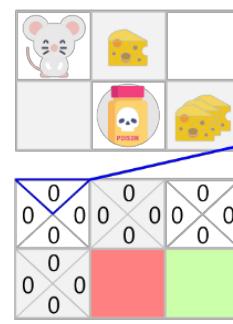
C.Sanmiguel Vila

	Actions	←	→	↑	↓
States		0	0	0	0
Mouse (Top Left)		0	0	0	0
Cheese (Top Middle)		0	0	0	0
Empty Space (Bottom Left)		0	0	0	0
Empty Space (Bottom Middle)		0	0	0	0
Empty Space (Bottom Right)		0	0	0	0
Poison Jar (Bottom Left)		0	0	0	0
Cheese (Bottom Middle)		0	0	0	0

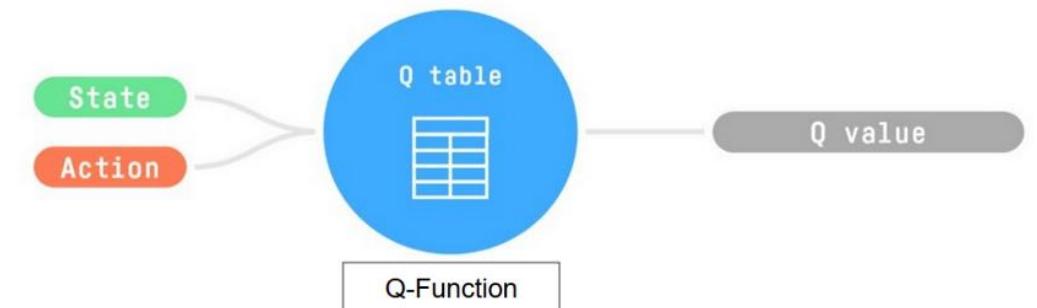
The RL Process

Here we see that the **state-action value of the initial state and going up is 0:**

So: the Q-function uses a Q-table **with the value of each state-action pair.** Given a state and action, **our Q-function will search inside its Q-table to output the value.**



	←	→	↑	↓
Mouse	0	0	0	0
Cheese	0	0	0	0
Poison	0	0	0	0
Empty	0	0	0	0



The RL Process

If we recap, ***Q-Learning is the RL algorithm that:***

- Trains a ***Q-function*** (an ***action-value function***), which internally is a ***Q-table*** that ***contains all the state-action pair values***.
- Given a state and action, our Q-function ***will search its Q-table for the corresponding value***.
- When the training is done, ***we have an optimal Q-function, which means we have optimal Q-table***.
- And if we ***have an optimal Q-function, we have an optimal policy*** since we ***know the best action to take at each state***.

The link between Value and Policy:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

Finding an optimal value function leads to having an optimal policy.

The RL Process

In the beginning, **our Q-table is useless since it gives arbitrary values for each state-action pair** (most of the time, we initialize the Q-table to 0). As the agent **explores the environment and we update the Q-table, it will give us a better and better approximation to the optimal policy.**



The Q-Learning algorithm

Q-Learning

Algorithm 14: Sarsamax (Q-Learning)

Input: policy π , positive integer $num_episodes$, small positive fraction α , GLIE $\{\epsilon_i\}$

Output: value function Q ($\approx q_\pi$ if $num_episodes$ is large enough)

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(terminal-state, \cdot) = 0$)

for $i \leftarrow 1$ **to** $num_episodes$ **do**

Step 1

$\epsilon \leftarrow \epsilon_i$

 Observe S_0

$t \leftarrow 0$

repeat

 Choose action A_t using policy derived from Q (e.g., ϵ -greedy) **Step 2**

 Take action A_t and observe R_{t+1}, S_{t+1} **Step 3**

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$ **Step 4**

$t \leftarrow t + 1$

until S_t is terminal;

end

return Q

The Q-Learning algorithm

Q-Learning, Step 1

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\text{terminal-state}, \cdot) = 0$)

	\leftarrow	\rightarrow	\uparrow	\downarrow
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

We initialize the Q-Table

The Q-Learning algorithm

Q-Learning, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



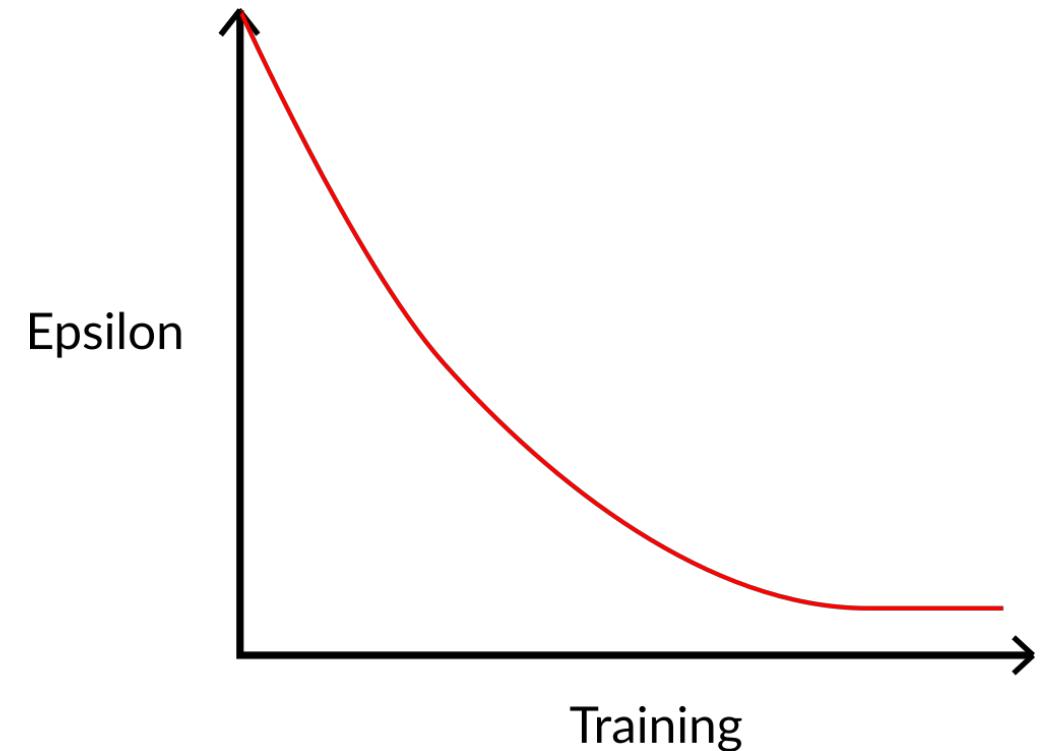
The Q-Learning algorithm

The epsilon-greedy strategy is a policy that handles the exploration/exploitation trade-off.

The idea is that, with an initial value of $\epsilon = 1.0$:

- *With probability $1 - \epsilon$: we do **exploitation** (aka our agent selects the action with the highest state-action pair value).*
- *With probability ϵ : we do **exploration** (trying random action).*

At the beginning of the training, **the probability of doing exploration will be huge since ϵ is very high, so most of the time, we'll explore**. But as the training goes on, and consequently our **Q-table gets better and better in its estimations, we progressively reduce the epsilon value** since we will need less and less exploration and more exploitation.



The Q-Learning algorithm

Step 3: Perform action A_t , get reward R_{t+1} and next state S_{t+1}

Q-Learning, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}

The Q-Learning algorithm

Step 4: Update $Q(S_t, A_t)$

Remember that in TD Learning, we update our policy or value function (depending on the RL method we choose) **after one step of the interaction**.

To produce our TD target, **we used the immediate reward R_{t+1} plus the discounted value of the next state**, computed by finding the action that maximizes the current Q-function at the next state. (We call that bootstrap).

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

New value
of state t

Former
estimation of
value of state
t

Learning Rate
Reward

Discounted value of next
state

TD Target

The Q-Learning algorithm

Step 4: Update $Q(S_t, A_t)$

Therefore, our $Q(S_t, A_t)$ update formula goes like this:

$$Q(S_t, A_t) \leftarrow \underbrace{Q(S_t, A_t)}_{\text{New Q-value estimation}} + \underbrace{\alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}}]}_{\begin{array}{c} \text{Learning Rate} \\ \text{Immediate Reward} \\ \text{Discounted Estimate optimal Q-value of next state} \end{array}} \underbrace{- \underbrace{\gamma \max_a Q(S_{t+1}, a)}_{\text{TD Target}}}_{\text{TD Error}}$$

The Q-Learning algorithm

Step 4: Update $Q(S_t, A_t)$

This means that to update our $Q(S_t, A_t)$:

- We need $S_t, A_t, R_{t+1}, S_{t+1}$.
- To update our Q-value at a given state-action pair, we use the TD target.

How do we form the TD target?

1. We obtain the reward R_{t+1} after taking the action A_t .
2. To get the **best state-action pair value** for the next state, we use a greedy policy to select the next best action. Note that this is not an epsilon-greedy policy, this will always take the action with the highest state-action value.

Then when the update of this Q-value is done, we start in a new state and select our action **using a epsilon-greedy policy again**.

This is why we say that Q Learning is an off-policy algorithm.

Off-policy vs On-policy

The difference is subtle:

- **Off-policy:** using a different policy for acting (inference) and updating (training).

For instance, with Q-Learning, the epsilon-greedy policy (acting policy), is different from the greedy policy that is used to select the best next-state action value to update our Q-value (updating policy).

Choose action A_t using policy derived from Q (e.g., ϵ -greedy) Acting Policy
$$\gamma \max_a Q(S_{t+1}, a)$$
 Updating policy

- **On-policy:** using the same policy for acting and updating.

For instance, with Sarsa, another value-based algorithm, **the epsilon-greedy policy selects the next state-action pair, not a greedy policy.**

```
Choose action  $A_0$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)  
 $t \leftarrow 0$   
repeat  
  Take action  $A_t$  and observe  $R_{t+1}, S_{t+1}$   
  Choose action  $A_{t+1}$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)  
   $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$ 
```

Epsilon Greedy Policy

Q-Learning example

- You're a mouse in this tiny maze. You always **start at the same starting point**.
- The goal is to eat the big pile of cheese at the **bottom right-hand corner** and avoid the poison. After all, who doesn't like cheese?
- The episode ends if we eat the poison, **eat the big pile of cheese**, or if we take more than five steps.
- The learning rate is 0.1
- The discount rate (gamma) is 0.99



Q-Learning example

The reward function goes like this:

- **+0**: Going to a state with no cheese in it.
- **+1**: Going to a state with a small cheese in it.
- **+10**: Going to the state with the big pile of cheese.
- **-10**: Going to the state with the poison and thus dying.
- **+0** If we take more than five steps.



To train our agent to have an optimal policy (so a policy that goes right, right, down), **we will use the Q-Learning algorithm**.

Q-Learning example

So, for now, **our Q-table is useless**; we need to train our Q-function using the Q-Learning algorithm.

Let's do it for 2 training timesteps.

Step 1: Initialize the Q-table

Example, Step 1

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\text{terminal-state}, \cdot) = 0$)

	⬅	➡	↑	↓
🐭	0	0	0	0
🧀	0	0	0	0
	0	0	0	0
	0	0	0	0
🍯	0	0	0	0
🧀	0	0	0	0

We initialize the Q-Table

Q-Learning example

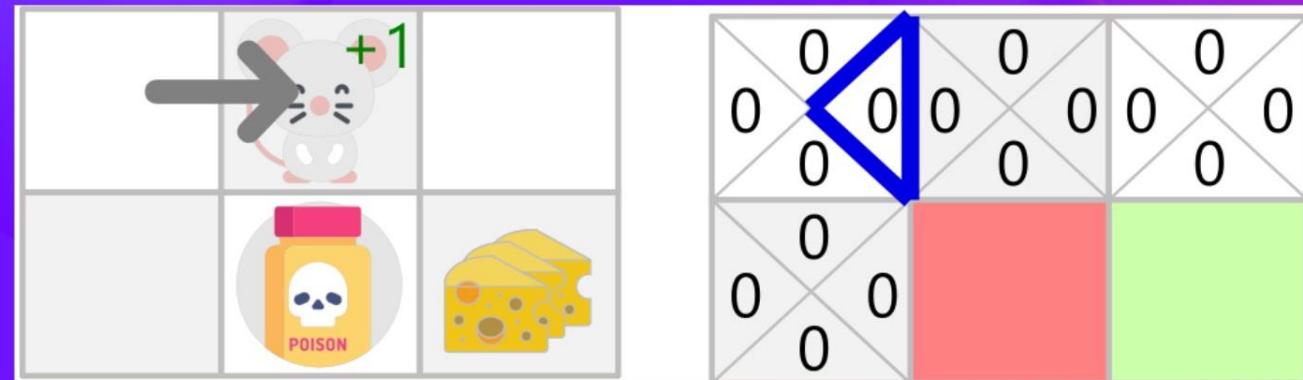
Training timestep 1:

Because epsilon is big (= 1.0), I take a random action. In this case, I go right.

Step 2: Choose an action using the Epsilon Greedy Strategy

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Q-Learning example

Training timestep 1:

By going right, I get a small cheese, so $R_{t+1} = 1$ and I'm in a new state.

Step 3: Perform action A_t , get R_{t+1} and S_{t+1}

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Q-Learning example

Training timestep 1:

Step 4: Update $Q(S_t, A_t)$

We can now update $Q(S_t, A_t)$ using our formula.

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

New Q-value estimation

Former Q-value estimation

Learning Rate

Immediate Reward

Discounted Estimate optimal Q-value of next state

Former Q-value estimation

TD Target

TD Error

Update our Q-value estimation

Q-Learning example

Training timestep 1:

We can now update $Q(S_t, A_t)$ using our formula.

Step 4: Update $Q(S_t, A_t)$

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

$$Q(\text{Initial state, Right}) = 0 + 0.1 * [1 + 0.99 * 0 - 0]$$

$$Q(\text{Initial state, Right}) = 0.1$$

	←	→	↑	↓
Initial state	0	0.1	0	0
Yellow block	0	0	0	0
Empty space	0	0	0	0
Grey block	0	0	0	0
Red block	0	0	0	0
Blue block	0	0	0	0

Q-Learning example

Training timestep 2:

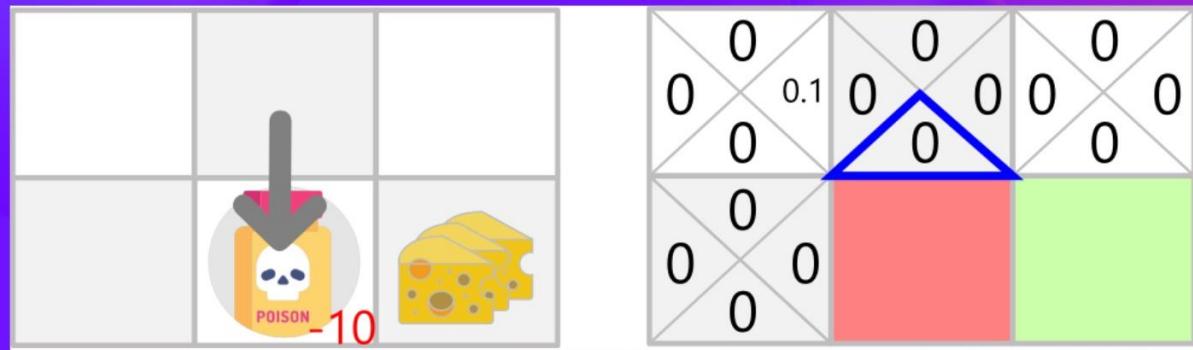
We take a random action again, since $\epsilon = 0.99$ is big. (Notice we decay epsilon a little bit because, as the training progress, we want less and less exploration).

I took the action 'down'. This is not a good action since it leads me to the poison.

Step 2: Choose an action using the Epsilon Greedy Strategy

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Q-Learning example

Training timestep 2:

Because we ate poison, I get $R_{t+1}=-10$, and we die.

Step 3: Perform action A_t , get R_{t+1} and S_{t+1}

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Q-Learning example

Training timestep 2:

We can now update $Q(S_t, A_t)$ using our formula.

Step 4: Update $Q(S_t, A_t)$

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

$$\begin{aligned} Q(\text{State 2, Down}) &= 0 + 0.1 * [-10 + 0.99 * 0 - 0] \\ Q(\text{State 2, Down}) &= -1 \end{aligned}$$

	←	→	↑	↓
Mouse	0	0.1	0	0
Hamster	0	0	0	-1
Blank	0	0	0	0
Blank	0	0	0	0
Peanut	0	0	0	0
Cheese	0	0	0	0

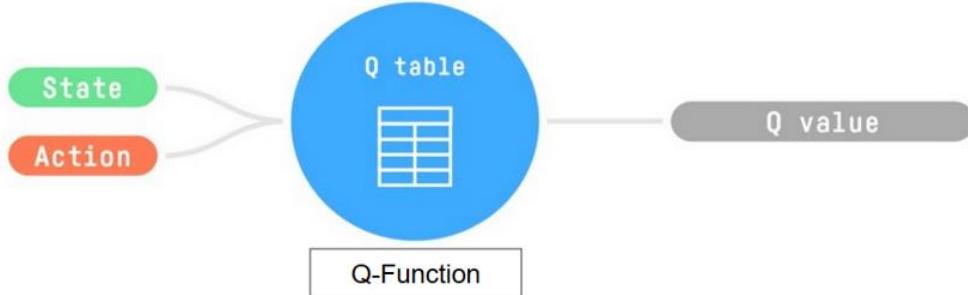
Q-Learning example

Because we're dead, we start a new episode. But what we see here is that, **with two explorations steps, my agent became smarter.**

As we continue exploring and exploiting the environment and updating Q-values using the TD target, the **Q-table will give us a better and better approximation. At the end of the training, we'll get an estimate of the optimal Q-function.**

Q-Learning Recap

Q-Learning is the RL algorithm that :



- Trains a *Q-function*, an **action-value function** encoded, in internal memory, by a *Q-table* **containing all the state-action pair values**.
- Given a state and action, our *Q-function* **will search its *Q-table* for the corresponding value**.
- When the training is done, **we have an optimal *Q-function*, or, equivalently, an optimal *Q-table***.
- And if we **have an optimal *Q-function***, we have an optimal policy, since we **know, for each state, the best action to take**.

Q-Learning Recap

But, in the beginning, our **Q-table** is useless since it gives arbitrary values for each state-action pair (most of the time we initialize the Q-table to 0 values).

However, as we explore the environment and update our Q-table, we will see a better approximation.

The link between Value and Policy:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

Finding an optimal value function leads to having an optimal policy.

Q-Learning

Wall	0	0	0	0
Gold	0	0	0	0
Empty	0	0	0	0
Empty	0	0	0	0
Bomb	0	0	0	0
Bomb	0	0	0	0

Training

Wall	0	10.8	0	0
Gold	0	9.9	0	-10
Empty	0	0	0	10
Empty	0	-10	0	0
Bomb	0	0	0	0
Bomb	0	0	0	0

Q-Learning Recap

Strategies to find the optimal policy

Policy-based methods. The policy is usually trained with a neural network to select what action to take given a state. In this case it is the neural network which outputs the action that the agent should take instead of using a value function. Depending on the experience received by the environment, the neural network will be re-adjusted and will provide better actions.

Value-based methods. In this case, a value function is trained to output the value of a state or a state-action pair that will represent our policy. However, this value doesn't define what action the agent should take. In contrast, we need to specify the behavior of the agent given the output of the value function. For example, we could decide to adopt a policy to take the action that always leads to the biggest reward (Greedy Policy). In summary, the policy is a Greedy Policy (or whatever decision the user takes) that uses the values of the value-function to decide the actions to take.

Q-Learning Recap

Among the value-based methods, we can find two main strategies

- **The state-value function.** For each state, the state-value function is the expected return if the agent starts in that state and follows the policy until the end.
- **The action-value function.** In contrast to the state-value function, the action-value calculates for each state and action pair the expected return if the agent starts in that state, takes that action, and then follows the policy forever after.

Q-Learning Recap

Epsilon-greedy strategy:

- A common strategy used in reinforcement learning that involves balancing exploration and exploitation.
- Chooses the action with the highest expected reward with a probability of 1-epsilon.
- Chooses a random action with a probability of epsilon.
- Epsilon typically decreases over time to shift focus towards exploitation.

Greedy strategy:

- Involves always choosing the action that is expected to lead to the highest reward, based on the current knowledge of the environment. (Only exploitation)
- Always choose the action with the highest expected reward.
- Does not include any exploration.
- Can be disadvantageous in environments with uncertainty or unknown optimal actions.

Q-Learning Recap

Off-policy vs on-policy algorithms

- **Off-policy algorithms:** A different policy is used at training time and inference time
- **On-policy algorithms:** The same policy is used during training and inference

Monte Carlo and Temporal Difference learning strategies

- **Monte Carlo (MC):** Learning at the end of the episode. With Monte Carlo, we wait until the episode ends and then we update the value function (or policy function) from a complete episode.
- **Temporal Difference (TD):** Learning at each step. With Temporal Difference Learning, we update the value function (or policy function) at each step without requiring a complete episode.