Topics on Artificial Intelligence & Machine Learning

D. T. McGuiness, PhD

B.Sc
Data Science II
Lecture Book

2025.WS

Data Science II
Lecture Book

D. T. McGuiness, PhD

MCI

Current version is 2025.WS.

This document includes the contents of Data Science II, official name being *Machine Learning and Data Science 2*, taught at MCI in the Mechatronik Design Innovation. This document is the part of the module MECH-B-5-MLDS-MLDS2-ILV taught in the B.Sc degree.

All relevant code of the document is done using *SageMath* where stated and Python v3.13.7.

This document was compiled with LuaTEX v1.22.0, and all editing were done using GNU Emacs v30.1 using AUCTEX and org-mode package.

This document is based on the following books and resources shown in no particular order:

Neural Networks: Methodology and Applications by Gérard Dreyfus , Springer *Python for Data Analysis: Data Wrangling with Pandas, Numpy, and iPython* by Wes McKinney , Springer *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* by Aurélien Géron , O'Reilly *TensorFlow for Deep Learning: From Linear Regression To Reinforcement Learning* by B. Ramsundar, and R. B. Zadeh , O'Reilly *AI and Machine Learning for Coders* by Moroney L. , O' Reilly *Neural Networks and Deep Learning* by Aggarwal S. , Springer *Python Machine Learning* by Raschka., et. al. , Packt *Machine Learning with Python Cookbook* by Albon C. , O' Reilly *CS229 Lecture Notes* by Ng A., et.al , - *Lecture Notes on Machine Learning* by Migel A., et. al , -

The document is designed with no intention of publication and has only been designed for education purposes.

The current maintainer of this work along with the primary lecturer

is D. T. McGuines, Ph.D. (dtm@mci4me.at).

Table of Contents

Chapter 7 Computer Vision using Convolutional Neural Networks

List of Figures

List of Tables

Part I

Machine Learning Algorithms

It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.

(Sir Arthur Conan Doyle, Sherlock Holmes)

Chapter 1

Support Vector Machines

Table of Contents

1.1 Introduction

One of the important task a ML has to accomplish is to classify data. An a useful method for this task is called SVM.

A powerful ML model, capable of performing linear or non-linear classification, regression. They are classified as supervised learning which utilises statistical learning[1] which can be used for pattern recognition and regression. It can also identify precisely the factors which need to be taken into account to learn successfully certain simple types of algorithms, however, real-world applications usually need more complex models and algorithms (such as neural networks), that makes them much harder to analyse theoretically.

SVMs can be seen as lying at the intersection of learning theory and practice. They construct models that are complex enough (containing a large class of neural networks for instance) and yet that are simple enough to be analysed mathematically.

> It is even possible to implement **novelty detection** using SVM

SVMs most suitable for small to medium sized non-linear datasets,[2] especially for classification tasks.

[1] A framework for machine learning drawing from the fields of statistics and functional analysis. Statistical learning theory deals with the statistical inference problem of finding a predictive function based on data. Statistical learning theory has led to successful applications in fields such as computer vision, speech recognition, and bio-informatics.

[2] i.e., hundreds to thousands of instances.

1.2 Linear Support Vector Machine Classification

As with most engineering and abstract concepts, the idea behind SVM is best explained using programming and some visuals. **Fig.** 1.1 shows part of the iris dataset[3] which was briefly discussed in **Data Science I**.

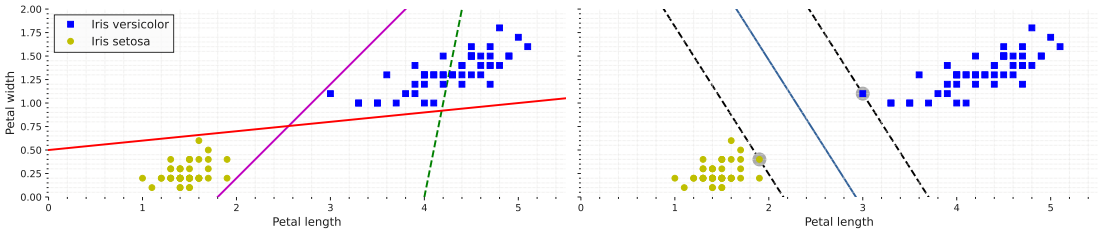


Figure 1.1: Large margin classification of the iris dataset.

The two (2) classes can easily and clearly separated with a straight line.

> This means the data is **linearly separable**.

The left plot shows the decision boundaries of three (3) possible linear classifiers. The model whose decision boundary is represented by the dashed line (- -) is so bad it does not even separate the classes properly.

The other two (2) models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably **NOT** perform as well on new instances.

In contrast, the black solid line in the plot on the right represents the **decision boundary** of an **SVM classifier**. This line not only separates the two classes but also stays as far away from the closest training instances as possible. Think of an SVM classifier as fitting the widest possible street, which in the plot is represented by the parallel dashed lines, between the classes.

This is called **large margin classification**.

Information: Margin Classifier

A classifier which is able to give an associated distance from the decision boundary for each example. For instance, if a linear classifier is used, the distance of an example from the separating hyperplane is the margin of that example.

Please observe that adding more training instances will **NOT** affect the decision boundary at all as the boundaries are fully determined[4] by the instances located on the edge of the boundaries. These instances are called the **support vectors**.

[3] The Iris flower data set or Fisher's Iris data set is a multivariate data set used and made famous by the British statistician and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis. It is sometimes called Anderson's Iris data set because Edgar Anderson collected the data to quantify the morphologic variation of Iris flowers of three related species.

[4] or supported.

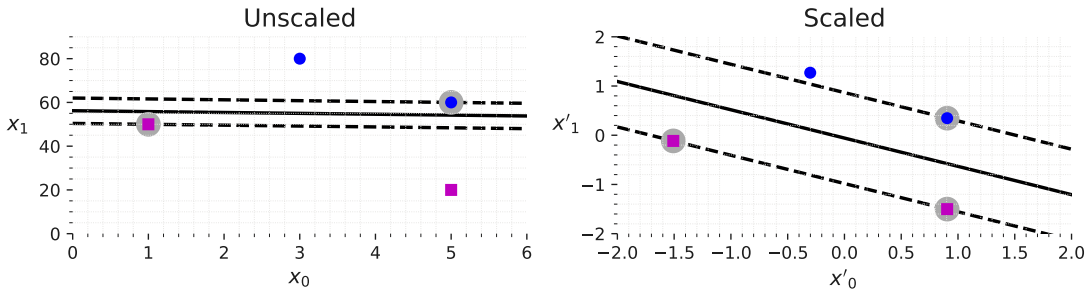


Figure 1.2: Sensitivity to feature scales.

SVMs are **sensitive to the feature scales**, as you can see in **Fig.** 1.2. In the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using `scikit-learn`'s `StandardScaler`), the decision boundary in the right plot looks much better.

1.2.1 Soft Margin Classification

If we impose all instances must be off the street and on the correct side, this is called **hard margin classification**. There are two (2) major issues with hard margin classification:

Sensitivity to Outliers
: Hard margin SVM is highly sensitive to outliers or noisy data points. Even a single mislabeled point can significantly affect the position of the decision boundary and lead to poor generalization on unseen data.

Not Suitable for Non-linear Data
: When the data is not linearly separable, hard margin SVM fails to find a valid solution, rendering it impractical for many real-world datasets.

Information: Outlier

A data point that differs significantly from other observations. An outlier may be due to a variability in the measurement, an indication of novel data, or it may be the result of experimental error.

Fig. 1.3 shows the iris dataset with just one (1) additional outlier. On the left, it is impossible to find a hard margin whereas on the right, the decision boundary ends up very different from the one we saw in **Fig.** 1.1 without the outlier, and the model will probably not generalize as well.

To avoid these aforementioned issues, we need to use a more flexible model. The idea is to find a good balance between keeping the street as large as possible and limiting the margin violations.[5]

This is called soft margin classification.

[5] i.e., instances that end up in the middle of the street or even on the wrong side.

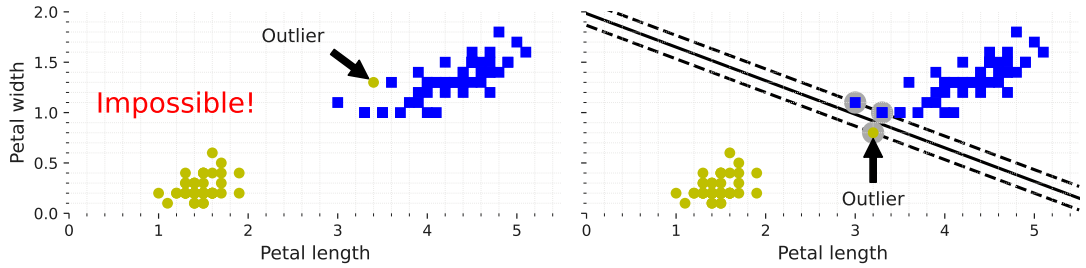


Figure 1.3: Hard margin sensitivity to outliers.

As mentioned, soft margin SVM introduces flexibility by allowing some margin violations, misclassifications, to handle cases where the data is **NOT** perfectly separable.

> This is suitable for scenarios where the data may contain noise or outliers.

It introduces a penalty term for misclassifications, allowing for a trade-off between a wider margin and a few misclassifications.

Soft margin SVM allows for some margin violations, meaning that it permits certain data points to fall within the margin or even on the wrong side of the decision boundary. This adaptability is managed by a factor called `C`, also called the "regularisation parameter", which helps find a balance between making the gap as big as possible and reducing mistakes in grouping things.

When creating an SVM model using `scikit-learn`, you can specify several hyperparameters, including the regularisation hyperparameter `C`.

Information: Hyperparameter `C`

Regularisation parameter. The strength of the regularisation is inversely proportional to `C` and must be **strictly positive**. The penalty is a squared ℓ_2 penalty.

Setting the `C` to a low value, we end up with the model on the left of **Fig.** 1.4. However, setting a high value, we get the model on the right. As can be seen, reducing `C` makes the street larger, but it also leads to more margin violations.

In other words, reducing `C` results in more instances supporting the street, so there's less risk of over-fitting. But if you reduce it too much, then the model ends up underfitting, as seems to be the case here:

The model with `C=100` looks like it will generalise better than the one with `C=1`.

> If your SVM model is overfitting, we can try regularizing it by reducing `C`.

The following `scikit-learn` code loads the iris dataset and trains a linear SVM classifier to detect *Iris virginica* flowers. The pipeline first scales the features, then uses a `LinearSVC` with `C=1`:

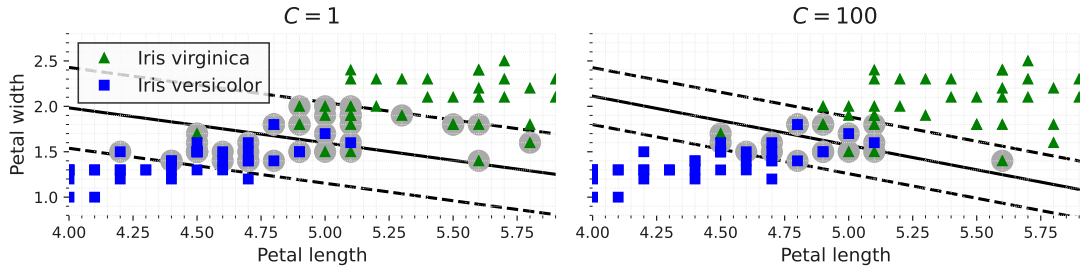


Figure 1.4: Large margin (left) v. fewer margin violations (right).

C.R. 1 python
```
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 2) # Iris virginica
svm_clf = make_pipeline(StandardScaler(),
LinearSVC(C=1, random_state=42))
svm_clf.fit(X, y)
```

The resulting model is given on the left in **Fig.** 1.4. Then, we can use the model to make predictions:

C.R. 2 python
```
X_new = [[5.5, 1.7], [5.0, 1.5]]
print(svm_clf.predict(X_new))
```

C.R. 3 text
```
[ True False]
```

The first plant is classified as an *iris virginica*, while the second is **NOT**. Let us look at the scores that the SVM used to make these predictions. These measure the signed distance between each instance and the decision boundary:

C.R. 4 python
```
print(svm_clf.decision_function(X_new))
```

C.R. 5 text
```
[ 0.66163816 -0.22035761]
```

Unlike the `LogisticRegression` class, `LinearSVC` doesn't have a `predict_proba()` method to estimate the class probabilities. That said, if you use the `SVC` class instead of `LinearSVC`, and if you set its probability hyperparameter to `True`, then the model will fit an extra model at the end of training to map the SVM decision function scores to estimated probabilities.

Under the hood, this requires using 5-fold cross-validation to create out-of-sample predictions for every instance in the training set, then training a `LogisticRegression` model, which will slow down training. After that, the `predict_proba()` and `predict_log_proba()` methods will be available.

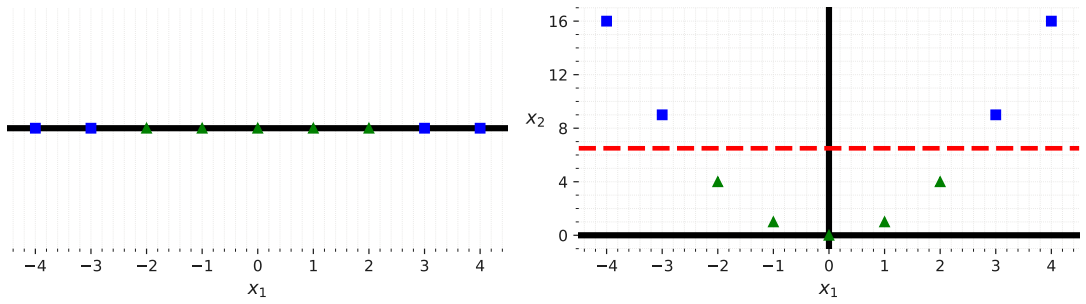


Figure 1.5: Adding features to make a dataset linearly separable.

1.3 Nonlinear Support Vector Machine Classification

Although linear SVM classifiers are efficient and often work surprisingly well, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features.

In some cases this can result in a linearly separable dataset. Consider the LHS plot in **Fig.** 1.5 where it represents a simple dataset with just one feature; x_1.

As we can see, this dataset is **NOT** linearly separable. But adding a second feature $x_2 = x_1^2$, the resulting 2D dataset is perfectly linearly separable.

To implement this idea using `scikit-learn`, we can create a pipeline containing a `PolynomialFeatures` transformer, followed by a `StandardScaler` and a `LinearSVC` classifier.

Information: Pipeline

A series of interconnected data processing and modeling steps for streamlining the process of working with ML models.

Let's test this on the moons dataset, a toy dataset for binary classification in which the data points are shaped as two (2) interleaving crescent moons.

We can generate this dataset using the `make_moons()` function, shown in **Fig.** 1.6.

C.R. 6
python
```python
from sklearn.datasets import make_moons
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15, random_state=42)
polynomial_svm_clf = make_pipeline(
    PolynomialFeatures(degree=3),
    StandardScaler(),
    LinearSVC(C=10, max_iter=10_000, random_state=42))
polynomial_svm_clf.fit(X, y)
```

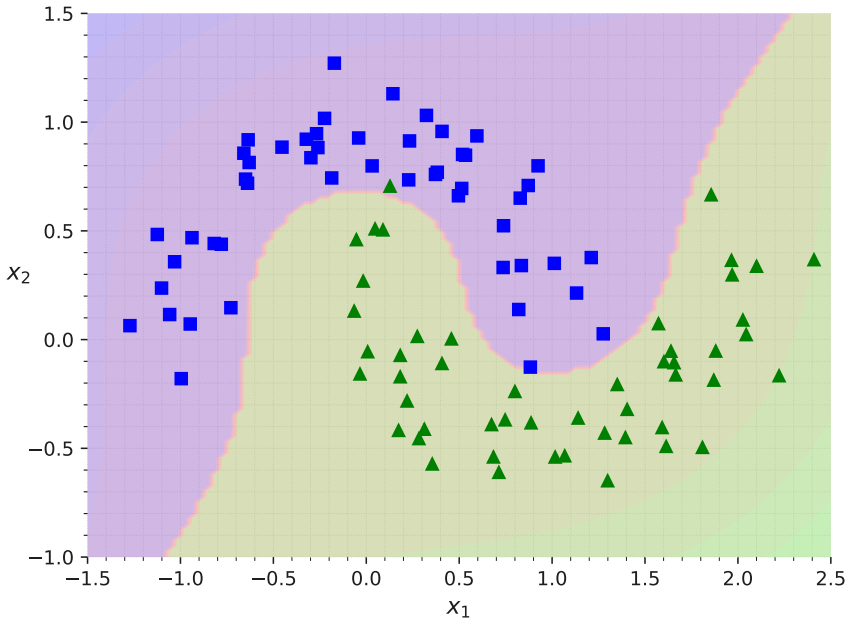


Figure 1.6: Linear support vector machine classifier using polynomial features.

1.3.1 Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of ML algorithms, and not just SVMs. That said, at a low polynomial degree this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply a mathematical technique called the kernel trick. The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with a very high degree, without actually having to add them.

> This means there's no combinatorial explosion of the number of features.

This trick is implemented by the `SVC` class. Let's test it on the moons dataset:

C.R. 7
python
```python
from sklearn.svm import SVC

poly_kernel_svm_clf = make_pipeline(
    StandardScaler(),
    SVC(kernel="poly", degree=3, coef0=1, C=5)
)
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a 3rd degree polynomial kernel, represented on the left in **Fig.** 1.7. On the right is another SVM classifier using a 10th degree polynomial kernel. Obviously,

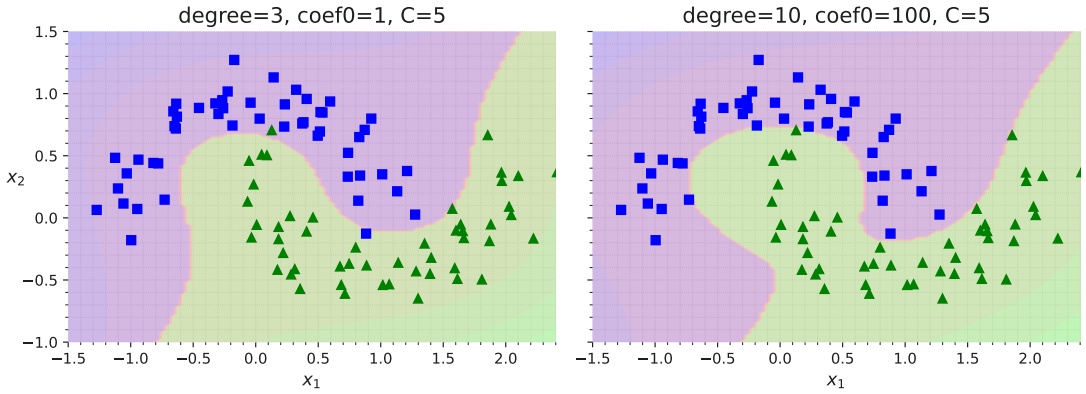


Figure 1.7: SVM classifiers with a polynomial kernel.

if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree terms versus low-degree terms.

> Although hyperparameters will generally be tuned automatically (e.g., using randomised search), it's good to have a sense of what each hyperparameter actually does and how it may interact with other hyperparameters: this way, you can narrow the search to a much smaller space.

1.3.2 The Kernel Trick

We have seen how higher dimensional transformations[6] can allow us to separate data in order to make classification predictions. It seems that in order to train a support vector classifier and optimize our objective function, we would have to perform operations with the higher dimensional vectors in the transformed feature space.

[6] i.e., our previous example.

In real applications, there might be many features in the data and applying transformations that involve many polynomial combinations of these features will lead to extremely high and impractical computational costs.

The kernel trick provides a solution to this problem. The **trick** is that kernel methods represent the data only through a set of pairwise similarity comparisons between the original data observations x,[7] instead of explicitly applying the transformations $\phi(x)$ and representing the data by these transformed coordinates in the higher dimensional feature space.

[7] with the original coordinates in the lower dimensional space.

In kernel methods, the data set $\boldsymbol{X}$ is represented by an $n \times n$ kernel matrix of pairwise similarity comparisons where the entries (i, j) are defined by the kernel function: $k\left(x_i, x_j\right)$.

This kernel function has a special mathematical property. The kernel function acts as a modified dot

product.

Theory 1.1: The Kernel Function

Kernel is defined as the function which takes as its inputs vectors in the original space and returns the dot product of the vectors in the feature space called the **kernel** function.

Formally, if we have data $\boldsymbol{x}$, $vz \in \boldsymbol{X}$ and a mapping of $\phi : \boldsymbol{X} \rightarrow \mathbb{R}^N$, then:

$$k(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi(x), \phi(z) \rangle$$

is a kernel function ■

Our kernel function accepts inputs in the original lower dimensional space and returns the dot product of the transformed vectors in the higher dimensional space. There are also theorems which guarantee the existence of such kernel functions under certain conditions.

The ultimate benefit of the kernel trick is that the objective function we are optimizing to fit the higher dimensional decision boundary only includes the dot product of the transformed feature vectors. Therefore, we can just substitute these dot product terms with the kernel function, and we don't even use $\phi(x)$.

1.3.3 Similarity Features

Another technique to tackle non-linear problems is to add features computed using a similarity function, which measures how much each instance resembles a particular landmark.

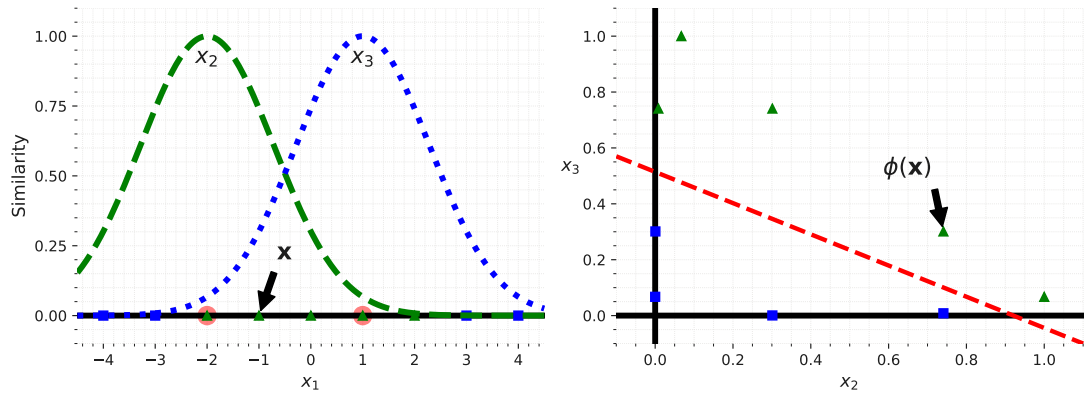


Figure 1.8: Similarity features using the Gaussian RBF.

For example, let's take the 1D dataset from earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in **Fig.** 1.8). Next, we'll define the similarity function to be the Gaussian Radial Basis Function (RBF) with $\gamma = 0.3$. As it is a Gaussian function, it is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark).

$$\phi_\gamma(\boldsymbol{x}, \ell) = \exp\left(-\gamma \mid\mid \boldsymbol{x} - \ell \mid\mid^2\right)$$

Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$. It is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore, its new features are:

$$x_2 = e^{-0.3 \times 1} = 0.75 \qquad x_3 = e^{-0.3 \times 4} = 0.3$$

The plot on the right in **Fig.** 1.8 shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. Doing that creates many dimensions and which increases the chances that the transformed training set will be linearly separable.

The downside is that a training set with m instances and n features gets transformed into a training set with m instances and m features.[8]

[8] assuming you drop the original features.

> A very large training set, ends up with an equally large number of features.

1.3.4 Gaussian RBF Kernel

Just like the polynomial features method, the similarity features method can be useful with any ML algorithm, but it may be computationally expensive to compute all the additional features.[9] Once again the kernel trick can be used here, making it possible to obtain a similar result as if you had added many similarity features, but without actually doing so. Let's try the `SVC` class with the Gaussian RBF kernel:

[9] especially on large training sets

C.R. 8

```python
rbf_kernel_svm_clf = make_pipeline(
    StandardScaler(),
    SVC(kernel="rbf", gamma=5, C=0.001)
)
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented at the bottom left in **Fig.** 1.9. The other plots show models trained with different values of hyperparameters gamma (γ) and `C`.

Increasing gamma makes the bell-shaped curve narrower (see the LHS plots in Figure 1.9). As a result, each instance's range of influence is **smaller**. The decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small gamma value makes the bell-shaped curve wider: instances have a larger range of influence, and the decision boundary ends up smoother.

Therefore γ acts like a **regularisation hyperparameter**: if your model is overfitting, you should reduce γ; if it is underfitting, you should increase γ (similar to the `C` hyperparameter).

Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. String kernels are sometimes used when classifying text documents or DNA sequences.[10]

[10] e.g., using the string subsequence kernel or kernels based on the Levenshtein distance.

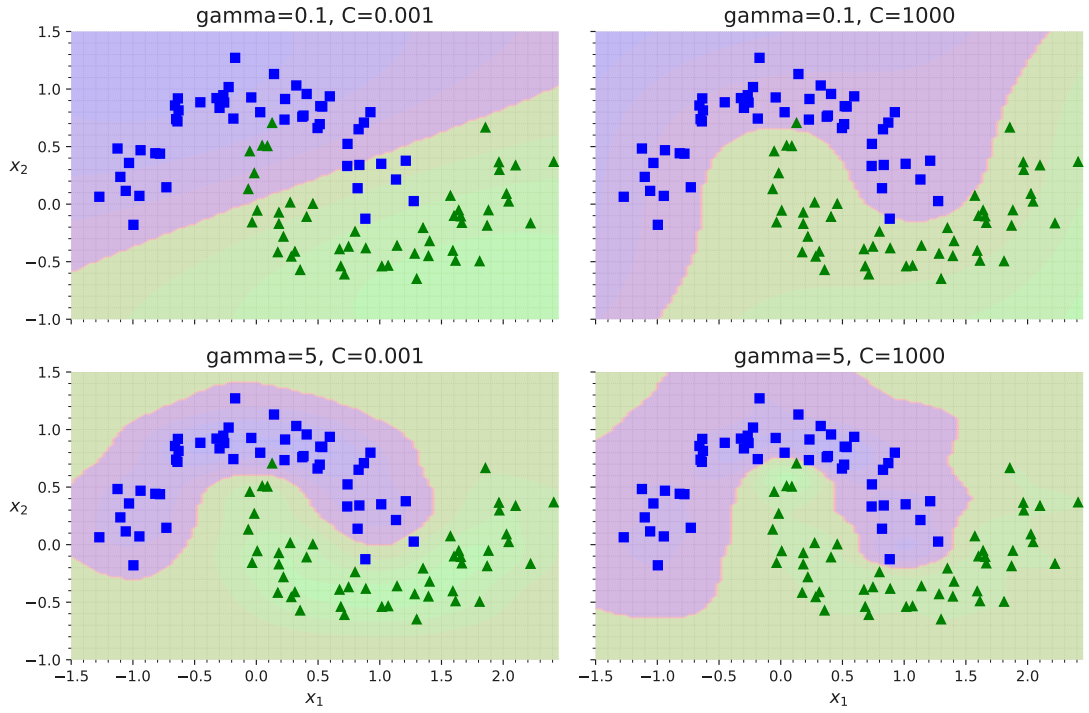


Figure 1.9: SVM classifiers using an RBF kernel.

You need to choose the right kernel for the job. As a rule of thumb, you should always try the linear kernel first. The `LinearSVC` class is much faster than `SVC(kernel="linear")`, especially if the training set is very large. If it is not too large, you should also try kernelised SVMs, starting with the Gaussian RBF kernel; it often works really well. Then, if you have spare time and computing power, you can experiment with a few other kernels using hyperparameter search.

If there are kernels specialized for your training set's data structure, make sure to give them a try too

1.4 Regression

To use SVMs for regression instead of classification, the main idea is to tweak the objective whereby instead of trying to fit the largest possible street between two (2) classes while limiting margin violations, SVM regression tries to fit as many instances as possible on the street while limiting margin violations.[11]

[11] i.e., instances off the street.

The width of the street is controlled by a hyperparameter, ϵ. **Fig.** 1.10 shows two (2) linear SVM regression models trained on some linear data, one with a small margin ($\epsilon = 0.5$) and the other with a larger margin ($\epsilon = 1.2$).

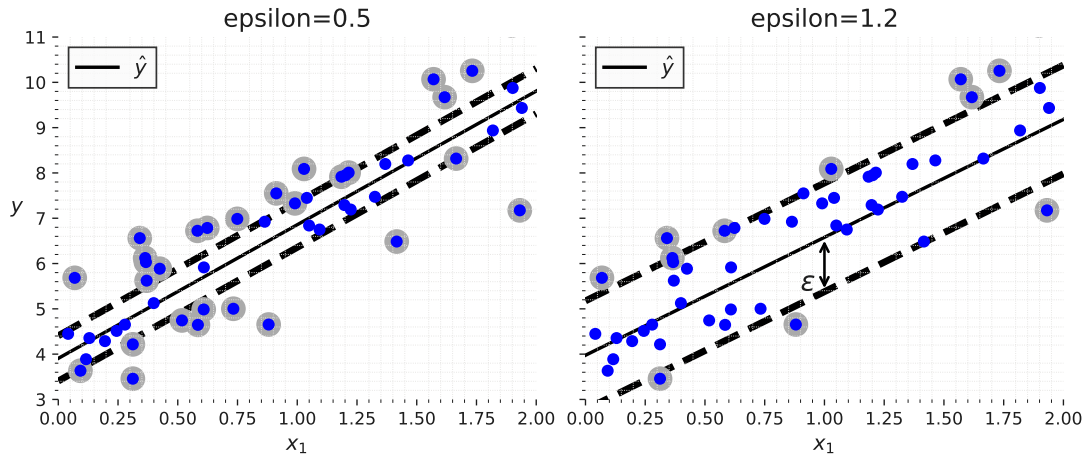


Figure 1.10: SVM regression.

Reducing ϵ increases the number of support vectors, which regularises the model. Moreover, if we were to add more training instances within the margin, it will not affect the model's predictions.

Therefore, the model is said to be ϵ-insensitive.

You can use `scikit-learn`'s `LinearSVR` class to perform linear SVM regression. The following code produces the model represented on the left in **Fig.** 1.10.

C.R. 9

```python
from sklearn.svm import LinearSVR

np.random.seed(42)
X = 2 * np.random.rand(50, 1)
y = 4 + 3 * X[:, 0] + np.random.randn(50)

svm_reg = make_pipeline(
    StandardScaler(),
    LinearSVR(epsilon=0.5, dual=True, random_state=42)
)
svm_reg.fit(X, y)
```

To tackle non-linear regression tasks, you can use a kernelized SVM model. **Fig.** 1.11 shows SVM

regression on a random quadratic training set, using a second-degree polynomial kernel. There is some regularisation in the left plot (i.e., a small `C` value), and much less in the right plot (i.e., a large `C` value).

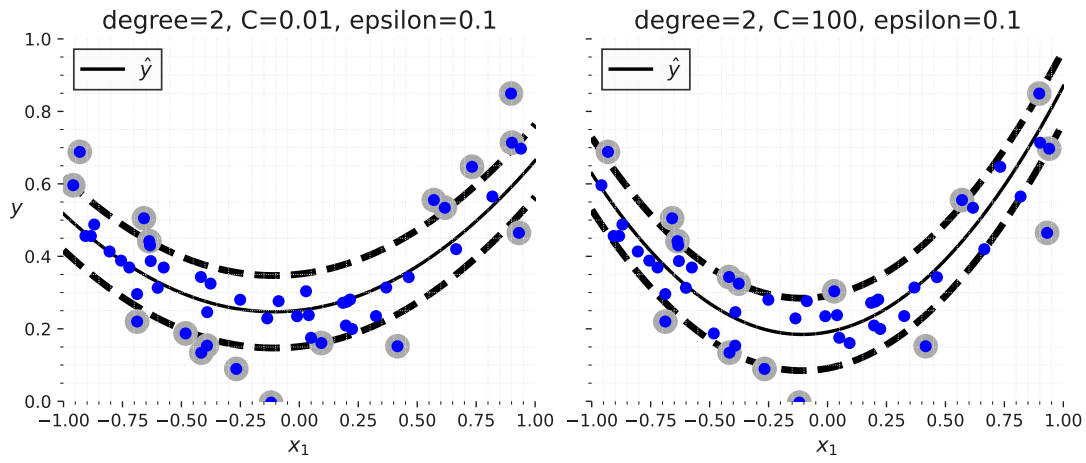


Figure 1.11: SVM regression using a second-degree polynomial kernel.

The following code uses `scikit-learn`'s `SVR` class[12] to produce the model represented on the left in **Fig.** 1.11:

[12] which supports the kernel trick

C.R. 10

```python
from sklearn.svm import SVR

# extra code  these 3 lines generate a simple quadratic dataset
np.random.seed(42)
X = 2 * np.random.rand(50, 1) - 1
y = 0.2 + 0.1 * X[:, 0] + 0.5 * X[:, 0] ** 2 + np.random.randn(50) / 10

svm_poly_reg = make_pipeline(
    StandardScaler(),
    SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1)
)

svm_poly_reg.fit(X, y)
```

The SVR class is the regression equivalent of the SVC class, and the `LinearSVR` class is the regression equivalent of the `LinearSVC` class. The `LinearSVR` class scales linearly with the size of the training set (just like the `LinearSVC` class), while the SVR class gets much too slow when the training set grows very large (just like the SVC class).

Chapter 2

Decision Trees

Table of Contents

2.1 Introduction

Decision Tree (DT) is a versatile ML algorithm which can perform both classification and regression tasks, and even multi-output tasks, capable of fitting complex datasets. They are classified as a non-parametric supervised learning algorithm, which is utilised for both classification and regression tasks.

It has a hierarchical tree structure, consisting of a root node, branches, internal nodes and leaf nodes. DTs are also the fundamental components of random forests, which are among the most powerful ML algorithms available today. Some of the applications include:

Loan Approval in Banking Banks use DTs to assess whether a loan application should be approved. The decision is based on factors such as credit score, income, and loan history. This helps predict approval or rejection and enables quick and reliable decisions [1], [2].

Medical Diagnosis In healthcare they assist in diagnosing diseases. For example, they can predict whether a patient has diabetes based on clinical data like glucose levels, Body-Mass Index (BMI) and blood pressure [3]. This helps classify patients into diabetic or non-diabetic categories,

supporting early diagnosis and treatment [4].

Predicting Exam Results in Education Educational institutions use to predict whether a student will pass or fail based on factors like attendance, study time and past grades. This helps teachers identify at-risk students and offer targeted support [5], [6].

Customer Churn Prediction Companies use DTs to predict whether a customer will leave or stay based on behaviour patterns, purchase history, and interactions. This allows businesses to take proactive steps to retain customers [7].

Fraud Detection In finance, DTs are used to detect fraudulent activities, such as credit card fraud [8]. By analysing past transaction data and patterns, DTs can identify suspicious activities and flag them for further investigation [9].

In this chapter we will start by discussing how to train, visualise, and make predictions with DTs. Then we will go through the Classification and Regression Tree (CART) training algorithm used by `scikit-learn`, and we will explore how to regularise trees and use them for regression tasks.

2.1.1 Advantages and Disadvantages

Before we start with our chapter on DT, lets list down the advantages it has over other methods:

Easy to Understand
DTs are visual which makes it easy to follow the decision-making process

Versatility
Can be used for both classification and regression problems.

No Need for Feature Scaling
Unlike many ML models, it doesn't require us to scale or normalise our data.

Handles Non-linear Relationships
It capture complex, non-linear relationships between features and outcomes effectively.

Interpretability
The tree structure is easy to interpret helps in allowing users to understand the reasoning behind each decision.

Handles Missing Data
It can handle missing values by using strategies like assigning the most common value or ignoring missing data during splits.

Of course, as with every method, there are it's disadvantages which are:

Over-fitting They can over-fit the training data if they are too deep which means they memorise

the data instead of learning general patterns. This leads to poor performance on unseen data.

Instability It can be unstable which means that small changes in the data may lead to significant differences in the tree structure and predictions.

Bias towards Features with Many Categories It can become biased toward features with many distinct values which focuses too much on them and potentially missing other important features which can reduce prediction accuracy.

Difficulty in Capturing Complex Interactions DTs may struggle to capture complex interactions between features which helps in making them less effective for certain types of data.

Computationally Expensive for Large Datasets For large datasets, building and pruning a DT can be computationally intensive, especially as the tree depth increases.

Information: White v. Black Box

DTs are intuitive, and their decisions are easy to interpret. Such models are often called **white box** models. In contrast, random forests and neural networks are generally considered **black box** models. They make great predictions, and we can easily check the calculations that they performed to make these predictions, however, it is usually hard to explain in simple terms why the predictions were made.

For example, if a neural network says that a particular person appears in a picture, it is hard to know what contributed to this prediction: Did the model recognise that person's eyes? Their mouth? Their nose? Their shoes? Or even the couch that they were sitting on? Conversely, DTs provide nice, simple classification rules that can even be applied manually if need be (e.g., for flower classification). The field of interpretable ML aims at creating ML systems that can explain their decisions in a way humans can understand. This is important in many domains-for example, to ensure the system does not make unfair decisions.

2.2 Training and Visualising Decision Trees

To understand DTs, let's build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the iris dataset:

C.R. 1 python
```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

We can visualise the trained DT by first using the `export_graphviz()` function to output a graph definition file called `iris_tree.dot`:

C.R. 2 python
```python
from sklearn.tree import export_graphviz
export_graphviz(
tree_clf,
out_file="iris_tree.dot",
feature_names=["petal length (cm)", "petal width (cm)"],
class_names=iris.target_names,
rounded=True,
filled=True
)
```

If we are using a Jupyter Notebook to study, we can use `graphviz.Source.from_file()` to load and display the file inline such as given below:

C.R. 3 python
```python
from graphviz import Source
Source.from_file("iris_tree.dot")
```

Information: Graphviz & DOT

Graphviz (short for Graph Visualisation Software) is a package of open-source tools for creating graphs. It takes text input in DOT format, generates images.

DOT is a graph description language. DOT files are usually with `.gv` filename extension.

2.3 Making Predictions

Let's see how the tree represented in **Fig.** 2.1 makes predictions.

Suppose we find an iris flower and want to classify it based on its petals. Looking at our three in **Fig.** 2.1, we start at the **root node**, which is depth 0, at the top. This node asks whether the flower's petal length is smaller than 2,45 cm. If it is, then we move down to the root's **left child node**, which is in this case it is depth 1, left, which is a **leaf node**, as in it does not have any child nodes, so it does not ask any questions as it simply look at the predicted class for that node, and the DT predicts that our flower is an *Iris setosa* (`class=setosa`).

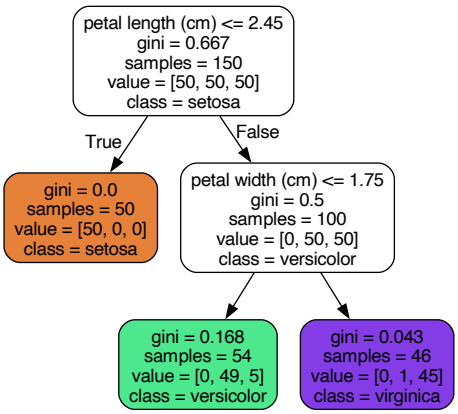


Figure 2.1: The iris decision tree.

Now suppose we find another flower, and this time the petal length is greater than 2,45 cm. We again start at the root but now move down to its right child node[1]. This is **NOT** a leaf node, it's a **split node**, so it asks another question:

[1] which is depth 1, right

> is the petal width smaller than 1,75 cm?

If it is, then our flower is most likely an *Iris versicolor*. If not, it is likely an *Iris virginica*.

> One of the many qualities of DTs is that they require very little data preparation. In fact, they don't require feature scaling or centring at all.

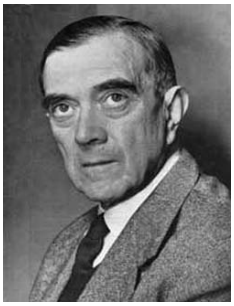
A node's samples attribute counts how many training instances it applies to.

> For example, 100 training instances have a petal length greater than 2,45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1,75 cm (depth 2, left).

A node's value attribute tells us how many training instances of each class this node applies to.

For example, the bottom-right node applies to 0 Iris setosa, 1 Iris versicolor, and 45 Iris virginica.

Finally, a node's gini attribute measures its **Gini impurity**, named after Corrado Gini[2] which we will have a look at now.



[2] **Corrado Gini** (*1884 - 1965*)

An Italian statistician, demographer and sociologist who developed the Gini coefficient, a measure of the income inequality in a society. Gini was a proponent of organicism and applied it to nations.

2.3.1 Gini Impurity

Gini Impurity is a measurement used to build DTs to determine how the features of a dataset should split nodes to form the tree. More precisely, the Gini Impurity of a dataset is a number between 0 - 0.5, which indicates the likelihood of new, random data being misclassified if it were given a random class label according to the class distribution in the dataset.

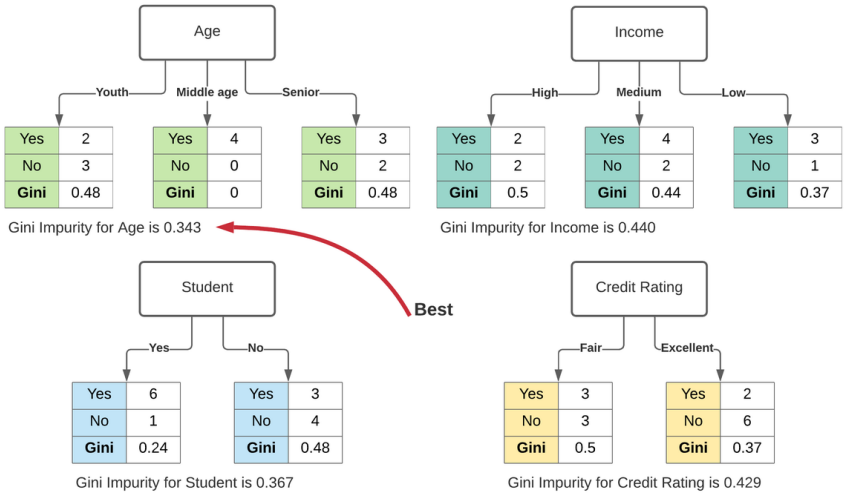


Figure 2.2: A visual description of Gini impurity.

For example, say we want to build a classifier which determines if someone will default on their credit card. We have some labelled data with features, such as bins for age, income, credit rating, and whether or not each person is a student. For us to find the best feature for the first split of the tree[3] we could calculate how poorly each feature divided the data into the correct class:

[3] i.e., the root node

- default ("yes"), or
- didn't default ("no").

This calculation would measure the impurity of the split, and the feature with the lowest impurity would determine the best feature for splitting the current node. This process would continue for each subsequent node using the remaining features.

In **Fig.** 2.2, age has minimum Gini impurity, so age is selected as the root in the decision tree.

> A node is said to be **pure** (`gini=0`) if all training instances belong to the same class.

Going back to our *iris* flower example, since the depth-1 left node applies only to *Iris setosa* training instances, it is **pure** and its Gini impurity is 0. Eq. (2.1) shows how the training algorithm computes the Gini impurity G_i of the i^{th} node. The depth-2 left node has a Gini impurity of

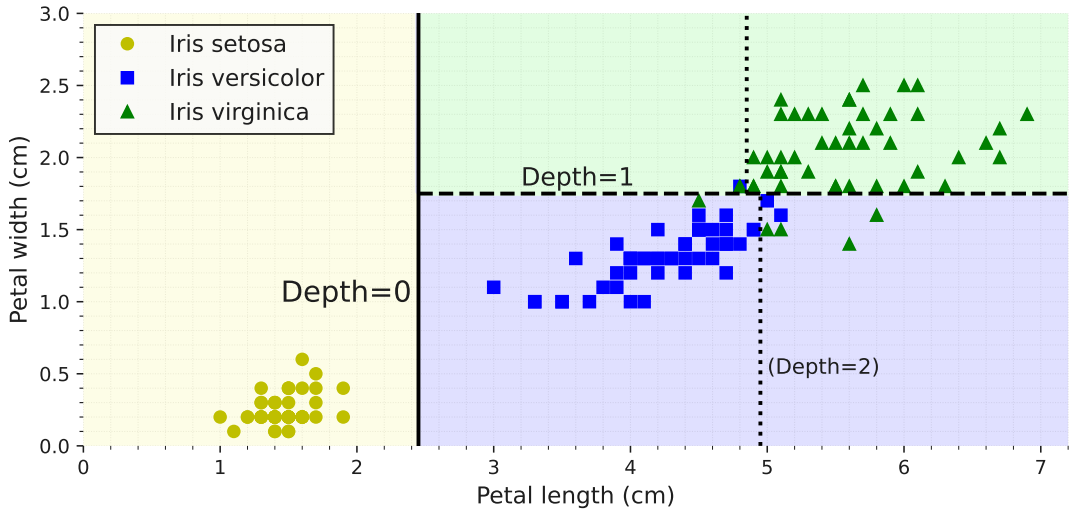


Figure 2.3: Decision tree decision boundaries.

Mathematically we can define the Gini impurity as:

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{2.1}$$

where G_i is the Gini impurity of the i^{th} node, $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node. Using this definition we can calculate the depth-2 left node as:

$$1 - \left(\frac{0}{54}\right)^2 - \left(\frac{49}{54}\right)^2 - \left((5)\,54\right)^2 \approx 0.168.$$

`scikit-learn` uses the CART algorithm, which produces only **binary trees**, meaning trees where split nodes always have exactly two (2) children.[4]

> However, other algorithms, such as ID3,[5] can produce DTs with nodes that have more than two children.

Fig. 2.3 shows this DT's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0) with petal length = 2,45 cm. Given the left hand area is pure (only *Iris setosa*), it cannot be split any further. However, the right hand area is **impure**, so the depth-1 right node splits it at petal width = 1,75 cm, which is represented by the dashed line.

Given `max_depth` was set to 2, the DT stops right there. If we set `max_depth` to 3, then the two depth-2 nodes would each add another decision boundary.[6]

> The tree structure, including all the information shown in Figure 2.1, is available via the classifier's `tree_` attribute. For more information, type `help(tree_clf.tree_)`.

[4] i.e., questions only have yes/no answers.

[5] ID3 (Iterative Dichotomiser 3) is an algorithm invented by Ross Quinlan used to generate a decision tree from a dataset, generally used in natural language processing.

[6] Which is represented by the two vertical dotted lines.

2.3.2 Estimating Class Probabilities

A DT can also estimate the probability which an instance belongs to a particular class k. First, it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class k in this node.

For example, suppose we have found a flower whose petals are 5 cm long and 1,5 cm wide. The corresponding leaf node is the depth-2 left node, so the DT outputs the following probabilities:

- 0% for Iris setosa (0/54),
- 90.7% for Iris versicolor (49/54), and
- 9.3% for Iris virginica (5/54).

If we ask it to predict the class, it outputs Iris versicolor (class 1) because it has the highest probability. Let's check this:

C.R. 4 python
```python
print(tree_clf.predict_proba([[5, 1.5]]).round(3))
print(tree_clf.predict([[5, 1.5]]))
```

C.R. 5 text
```
[[0.    0.907 0.093]]
[1]
```

Notice the estimated probabilities would be identical anywhere else in the bottom-right rectangle of **Fig.** 2.1, for example, if the petals were 6 cm long and 1,5 cm wide.

2.4 The CART Training Algorithm

CART is a predictive algorithm used for explaining how the target variable's values can be predicted based on other matters. It is a DT where each fork is split into a predictor variable and each node has a prediction for the target variable at the end. The three (3) primary points of the algorithm is as follows:

Tree structure

CART builds a tree-like structure consisting of nodes and branches. The nodes represent different decision points, and the branches represent the possible outcomes of those decisions. The leaf nodes in the tree contain a predicted class label or value for the target variable.

Splitting Criteria

CART uses a greedy approach to split the data at each node. It evaluates all possible splits and selects the one that best reduces the impurity of the resulting subsets. For classification tasks, CART uses Gini impurity as the splitting criterion. The lower the Gini impurity, the more pure the subset is. For regression tasks, CART uses residual reduction as the splitting criterion. The lower the residual reduction, the better the fit of the model to the data.

Pruning

To prevent over-fitting of the data, pruning is a technique used to remove the nodes that contribute little to the model accuracy. Cost complexity pruning and information gain pruning are two popular pruning techniques. Cost complexity pruning involves calculating the cost of each node and removing nodes that have a negative cost. Information gain pruning involves calculating the information gain of each node and removing nodes that have a low information gain.

`scikit-learn` uses the CART algorithm to train DTs (also called growing trees). The algorithm works by splitting the training set into two (2) subsets using a single feature k and a threshold t_k.[7]

[7] e.g., petal length $\leq$ 2,45 cm.

How does it choose k and t_k?

It searches for the pair $(k,\ t_k)$ that produces the purest subsets, weighted by their size. Eq. (2.2) gives the cost function that the algorithm tries to minimise.

$$J(k,\ t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}} \quad \text{where} \quad \begin{cases} G_{\text{left/right}} & \text{measures the impurity} \\ m_{\text{left/right}} & \text{number of instances} \end{cases} \tag{2.2}$$

Once the CART algorithm successfully splits the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops its recursive behaviour once it reaches the maximum depth,[8] or if it cannot find a split that will reduce impurity.

[8] defined by the `max_depth` hyperparameter.

A few other hyperparameters control additional stopping conditions:

`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, `max_leaf_nodes`.

CART algorithm is a **greedy algorithm**. This means it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not whether the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal.

2.4.1 Gini Impurity v. Information Entropy

By default, the `DecisionTreeClassifier` class uses the **Gini impurity** measure, but we can select the entropy impurity measure instead by setting the criterion hyperparameter to entropy.

Information: Entropy: A Measure of Disorder

The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered.

Entropy later spread to a wide variety of domains, including in Shannon's[9] information theory, where it measures the average information content of a message and the entropy is zero when all messages are identical.

In ML, entropy is frequently used as a measurement of impurity, a set's entropy is zero when it contains instances of only one class. Eq. (2.3) shows the definition of the entropy of the i^{th} node.

For example, the depth-2 left node in **Fig.** 2.1 has an entropy equal to:

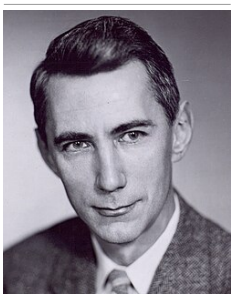
$$-\frac{49}{54}\log_2\frac{49}{54}-\frac{5}{54}\log_2\frac{5}{54}\approx 0.445$$

And the general equation for entropy could be written as:

$$H_{\text{i}}=-\sum_{k=1}^{n}p_{\text{i,k}}\log_2 p_{\text{i,k}} \qquad \text{where } p_{\text{i,k}}\neq 0 \tag{2.3}$$

So, which one to use? Gini impurity or entropy?

Most of the time it does not make a big difference as they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.



[9] **Claude Elwood Shannon** *(1916 - 2001)*

An American mathematician, electrical engineer, computer scientist, cryptographer and inventor known as the "father of information theory" and the man who laid the foundations of the Information Age. Shannon was the first to describe the use of Boolean algebra-essential to all digital electronic circuits-and helped found artificial intelligence (AI).

2.4.2 Regularisation Hyperparameters

DTs make very few assumptions about the training data.[10] If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely-indeed, resulting in **over-fitting**.

Such a model is often called a non-parametric model, not because it does not have any parameters[11]

[10] as opposed to linear models, which assume that the data is linear, for example

[11] as it often has a lot.

but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data.

In contrast, a parametric model, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of over-fitting.

> This, however, increases the risk of under-fitting.

To avoid over-fitting the training data, we need to restrict the DT's freedom during training. As we should know by now, this is called **regularisation**.

The regularisation hyperparameters depend on the algorithm used, but generally we can at least restrict the maximum depth of the DT. In `scikit-learn`, this is controlled by the `max_depth` hyperparameter. The default value is `None`, which means unlimited. Reducing `max_depth` will regularise the model and thus reduce the risk of over-fitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the DT:

`max_features` Maximum number of features that are evaluated for splitting at each node

`max_leaf_nodes` Maximum number of leaf nodes

`min_samples_split` Minimum number of samples a node must have before it can be split

`min_samples_leaf` Minimum number of samples a leaf node must have to be created

`min_weight_fraction_leaf` Same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances.

> Increasing `min_*` or reducing `max_*` hyperparameters will regularise the model.

Information: Other Methods of Training

Other algorithms work by first training the DT without restrictions, then pruning unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the χ^2 test (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the null hypothesis). If this probability, called the p-value, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Let's test regularisation on the moons dataset, introduced previously. We'll train one DT without regularisation, and another with `min_samples_leaf=5`. Here's the code with **Fig.** 2.4 showing the decision boundaries of each tree.

C.R. 6 python

```
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

tree_clf1 = DecisionTreeClassifier(random_state=42)
tree_clf2 = DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
tree_clf1.fit(X_moons, y_moons)
tree_clf2.fit(X_moons, y_moons)
```

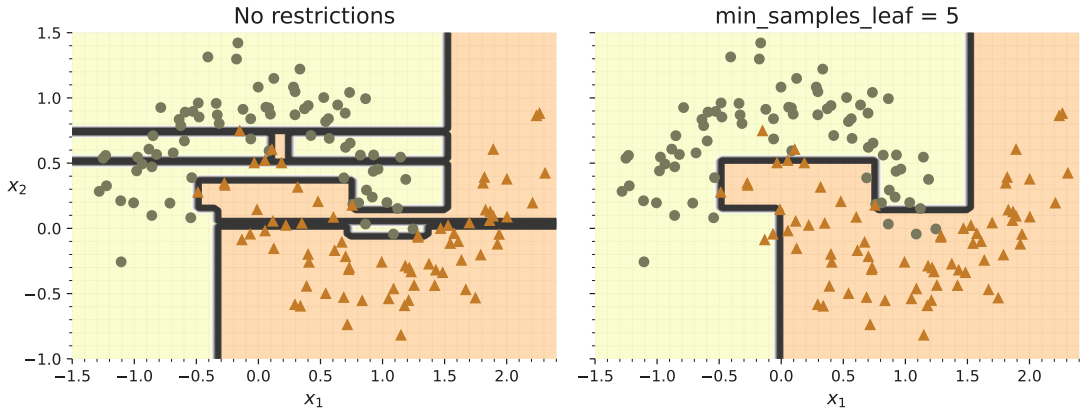


Figure 2.4: Decision boundaries of an unregulated tree (left) and a regularised tree (right).

The unregulated model on the left is clearly over-fitting, and the regularised model on the right will probably generalise better. We can verify this by evaluating both trees on a test set generated using a different random seed:

C.R. 7 python

```
X_moons_test, y_moons_test = make_moons(n_samples=1000, noise=0.2, random_state=43)
print(tree_clf1.score(X_moons_test, y_moons_test))
print(tree_clf2.score(X_moons_test, y_moons_test))
```

C.R. 8 text

```
0.898
0.92
```

As we can see, the 2nd tree has a better accuracy on the test set.

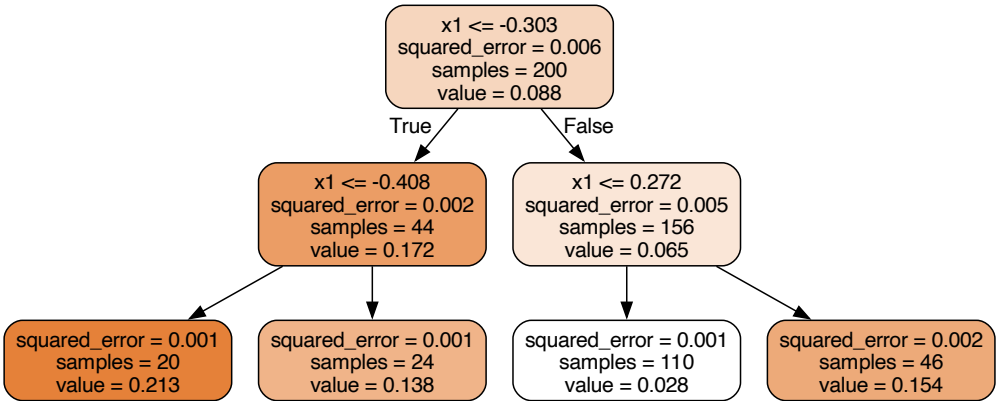


Figure 2.5: A decision tree for regression.

2.5 Regression

DTs are also capable of performing regression tasks. Let's build a regression tree with the `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2` with the resulting tree being represented in **Fig.** 2.5.

This tree looks very similar to the classification tree built earlier. The main difference is that instead of predicting a class in each node, it predicts a **value**.

For example, say we want to make a prediction for a new instance with $x_1 = 0.2$. The root node asks whether $x_1 \leq 0.197$. As it is not, the algorithm goes to the right child node, which asks whether $x_1 \leq 0.772$. Since it is, the algorithm goes to the left child node. This is a leaf node, and it predicts `value=0.111`. This prediction is the average target value of the 110 training instances associated with this leaf node, and results in a mean squared error equal to 0.015 over these 110 instances.

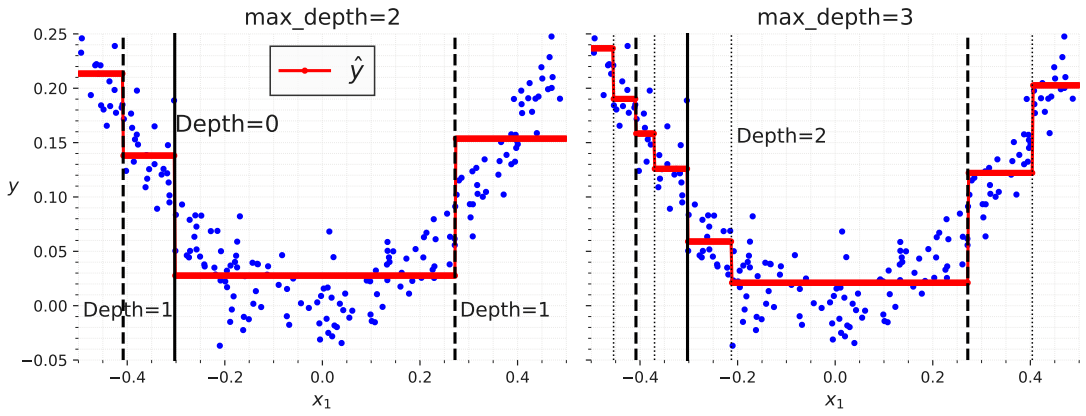


Figure 2.6: Predictions of two decision tree regression models.

This model's predictions are represented on the left in **Fig.** 2.6. If we set `max_depth=3`, we get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

The CART algorithm works as described earlier, except that instead of trying to split the training set in a way that minimises impurity, it now tries to split the training set in a way that minimises the Mean Square Error (MSE). To get a better feel of the underlying mathematics, Eq. (2.4) shows the cost function that the algorithm tries to minimise:

$$J\left(k,\, t_k\right) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \quad \begin{cases} \text{MSE}_{\text{node}} & = \sum\limits_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} & = \frac{1}{m_{\text{node}}} \sum\limits_{i \in \text{node}} y^{(i)} \end{cases} \tag{2.4}$$

Just like for classification tasks, DTs are prone to over-fitting when dealing with regression tasks. Without any regularization,[12] we get the predictions on the left in **Fig.** 2.7.

[12] i.e., using the default hyperparameters.

As can clearly be seen, these predictions are over-fitting the training set very badly. Setting `min_samples_leaf=10` results in a significantly more reasonable model, represented on the right in **Fig.** 2.7.

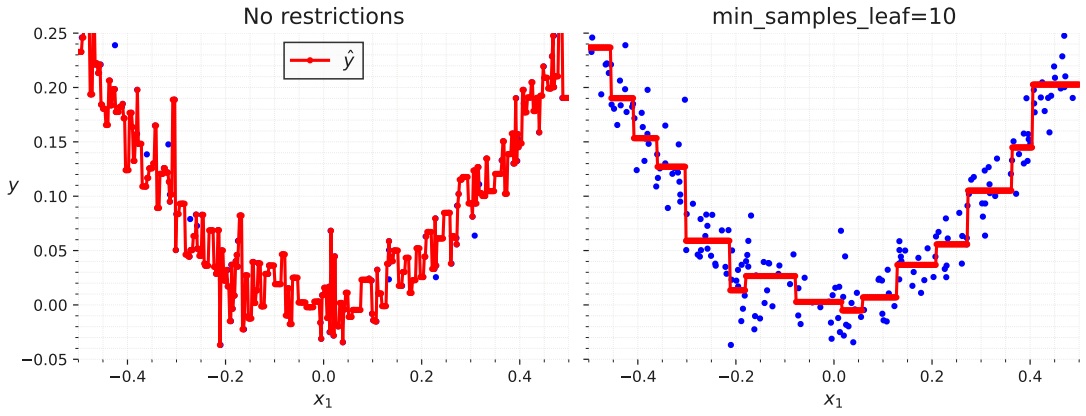


Figure 2.7: Predictions of an unregularised regression tree (left) and a regularised tree (right).

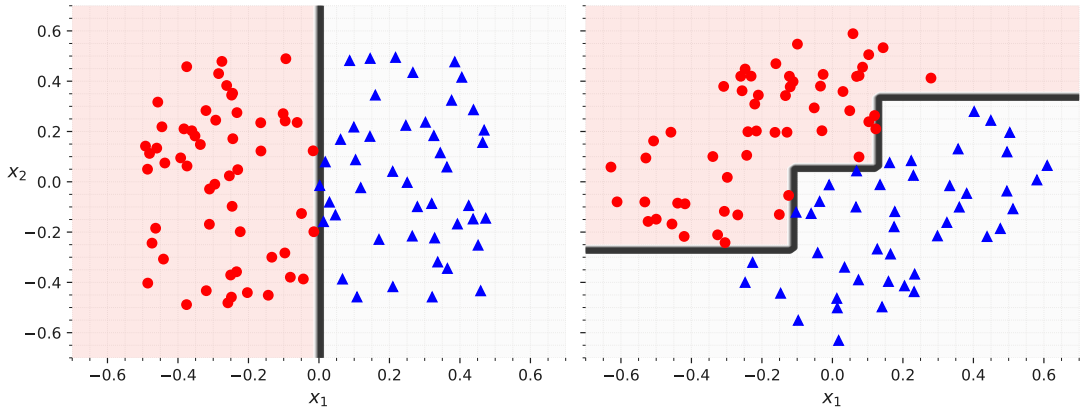


Figure 2.8: Sensitivity to training set rotation.

2.6 Sensitivity to Axis Orientation

DTs have a lot going for them given they are relatively easy to understand and interpret, simple to use, versatile, and powerful. However, they do have a few limitations. First, as we may have noticed, DTs love orthogonal decision boundaries,[13] which makes them sensitive to the orientation of data.

[13] all splits are perpendicular to an axis.

For example, **Fig.** 2.8 shows a simple linearly separable dataset where on the left, a DT can split it easily, while on the right, after the dataset is rotated by 45 degrees, the decision boundary looks unnecessarily convoluted. Although both DTs fit the training set perfectly, it is very likely that the model on the right will not generalise well.

One way to limit this problem is to scale the data, then apply a principal component analysis transformation. We will look at Principal Component Analysis (PCA) in detail later, but for now we only need to know that it rotates the data in a way that reduces the correlation between the features, which often makes things easier for trees.

Let's create a small pipeline that scales the data and rotates it using PCA, then, continue on to train a `DecisionTreeClassifier` on that data.

C.R. 9
python
```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris)
tree_clf_pca = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf_pca.fit(X_iris_rotated, y_iris)
```

Fig. 2.9 shows the decision boundaries of that tree and as we can see, the rotation makes it possible to fit the dataset pretty well using only one (1) feature, which is a linear function of the original petal length and width.

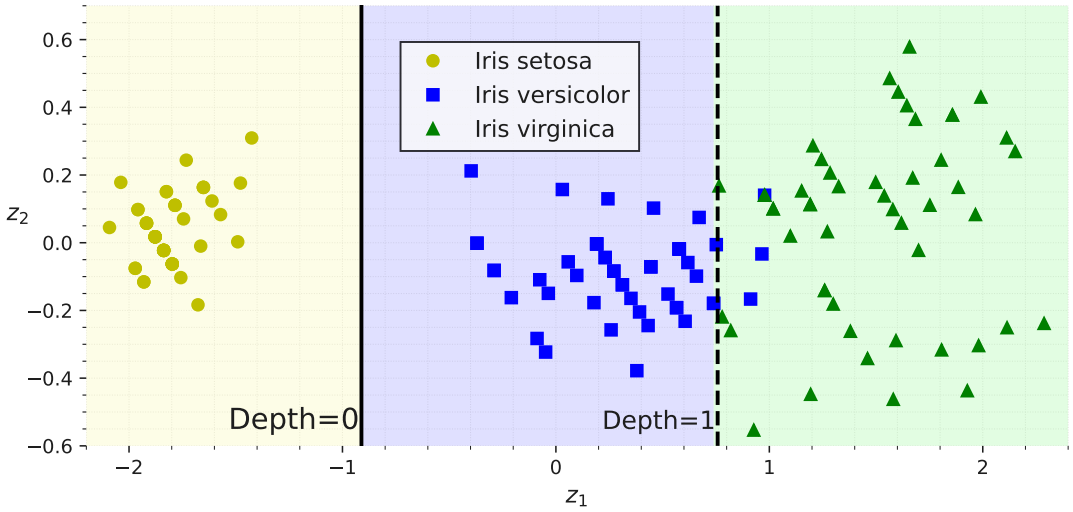


Figure 2.9: A tree's decision boundaries on the scaled and PCA-rotated iris dataset.

Information: The Problem of High Variance

More generally, the primary issue with DTs is that they have **high variance** where small changes to the hyperparameters or to the data may produce very different models.

In fact, given the training algorithm used by `scikit-learn` is stochastic-it randomly selects the set of features to evaluate at each node-even retraining the same DT on the exact same data may produce a very different model, such as the one represented in **Fig.** 2.10, unless we set the `random_state` hyperparameter.

As we can see, it looks very different from the previous DT, shown in **Fig.** 2.1.

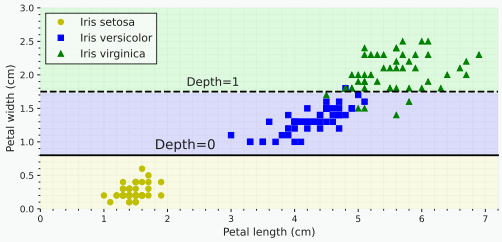


Figure 2.10: Retraining the same model on the same data may produce a very different model.

Luckily, by averaging predictions over many trees, it's possible to reduce variance significantly. Such an ensemble is called a RF, and it's one of the most powerful types of models available today.

Chapter 3

Ensemble Learning and Random Forests

Table of Contents

3.1 Introduction

Suppose you ask a complex question to millions of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer.

> This is called the wisdom of the crowd.[1]

In a similar fashion, aggregating the predictions of a group of predictors, such as classifiers or regressors. We will often get better predictions than with the best individual predictor.

> A group of predictors is called an **ensemble** and this technique, **ensemble learning**, and the learning method, **ensemble method**.

As an example of an ensemble method, we can train a group of decision tree classifiers, each on a

[1] also known as "wisdom of the majority", which expresses the notion that the collective opinion of a diverse and independent group of individuals (rather than that of a single expert) produces the best judgement [10].

different random subset of the training set. We can then obtain the predictions of all the individual trees, and the class that gets the most votes is the ensemble's prediction. Such an ensemble of decision trees is called a RF, and despite its simplicity, this is one of the most powerful ML algorithms available today.[2] In this chapter we will examine the most popular ensemble methods, including:

- voting classifiers,
- bagging and pasting ensembles,
- RFs,
- boosting,
- and stacking ensembles.

[2] **A Historical Overview**

The general method of random decision forests was originally proposed by Salzberg and Heath in 1993 [11], with a method that used a randomized decision tree algorithm to create multiple trees and then combine them using majority voting. This idea was developed further by Ho in 1995 [12]. Ho established that forests of trees splitting with oblique hyperplanes can gain accuracy as they grow without suffering from overtraining, as long as the forests are randomly restricted to be sensitive to only selected feature dimensions.

A subsequent work along the same lines [13] concluded that other splitting methods behave similarly, as long as they are randomly forced to be insensitive to some feature dimensions. This observation that a more complex classifier (a larger forest) gets more accurate nearly monotonically is in sharp contrast to the common belief that the complexity of a classifier can only grow to a certain level of accuracy before being hurt by overfitting.

3.1.1 Voting Classifiers

Let's assume that we have trained a few classifiers, with each one achieving about 80% accuracy.

> This may have a *logistic regression classifier*, an *SVM classifier*, a RF *classifier*, a *k-nearest neighbour classifier*, and perhaps a few more.

A simple way to create an even better classifier is to combine the predictions of each classifier where the class which gets the most votes is the ensemble's prediction.

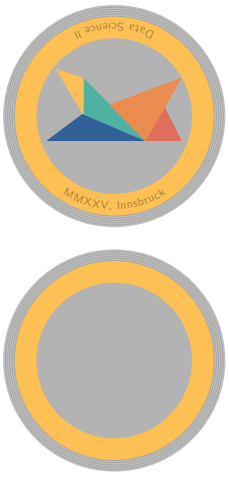
> This majority-vote classifier is called a **hard voting classifier**.

Interestingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a weak learner, meaning it does only slightly better than random guessing, the ensemble can still be a strong learner, achieving high accuracy, provided there are a sufficient number of weak learners in the ensemble and they are sufficiently diverse.

Let's discuss how this all works with an example. For simplicity, we will have a slightly biased coin which has a 51% chance of coming up heads and 49% chance of coming up tails.[3] If you toss it 1,000 times, we will generally get more or less 510 heads and 490 tails, and therefore will result in a majority of heads.

If you do the calculation, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the **law of large numbers**:

> as we keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%).



[3] A biased coin such as this, perhaps.

Information: Law of Large Numbers (LLN)

A mathematical law states that the average of the results obtained from a large number of independent random samples converges to the true value, if it exists. More formally, the LLN states that given a sample of independent and identically distributed values, the sample mean converges to the true mean. An example of this behaviour is shown in **Fig.** 3.1.

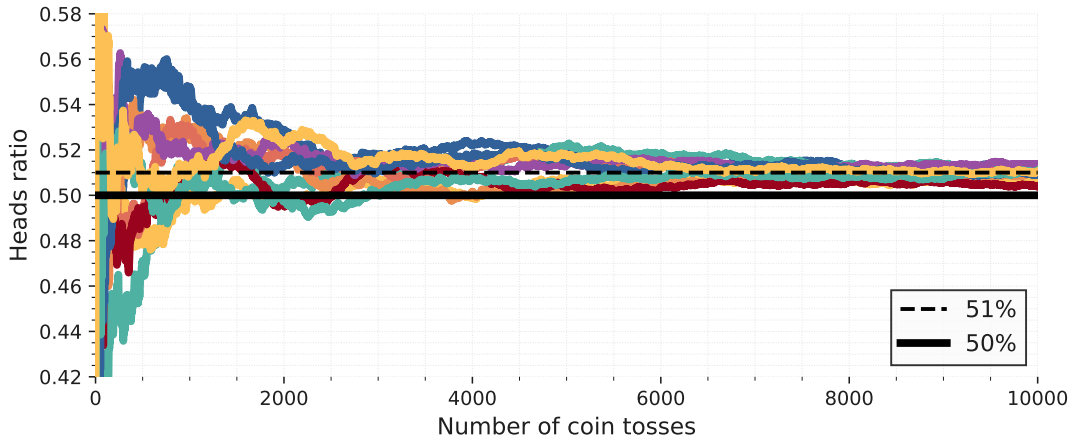


Figure 3.1: An example of a biased coin, where as the number of coin tosses increases the value of the will reach to 51%.

If we extend this analogy to our previous case, Our case becomes the building of an ensemble containing 1,000 classifiers which are individually correct only 51% of the time.[4]

[4] This can be barely considered to be better than just randomly guessing.

In this scenario, if we predict the majority voted class, we can hope for up to 75% accuracy.

However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy [14].

To implement this behaviour, `sklearn` provides a `VotingClassifier` class which is intuitive as it just gives it a list of name/predictor pairs, and use it like a normal classifier.

Let's try it on the moons dataset we used in SVM. We will load and split the moons dataset into a training set and a test set, then we'll create and train a voting classifier composed of three (3) diverse classifiers:

C.R. 1 python
```python
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
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)
voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

When we fit a `VotingClassifier`, it clones every estimator and fits the clones. The original estimators are available via the `estimators` attribute, whereas the fitted clones are available via the `estimators_` attribute.

> If we prefer a `dict` rather than a list, you can use `named_estimators` or `named_estimators_` instead.

To begin, let's look at each fitted classifier's **individual** accuracy on the test set:

C.R. 2 python
```python
for name, clf in voting_clf.named_estimators_.items():
    print(name, "=", clf.score(X_test, y_test))
```

C.R. 3 text
```
lr = 0.864
rf = 0.896
svc = 0.896
```

When we call the voting classifier's `predict()` method, it performs hard voting.

For example, the voting classifier predicts **class 1** for the first instance of the test set, because **two out of three classifiers** predict that class:

C.R. 4 python
```python
print(voting_clf.predict(X_test[:1]))
print([clf.predict(X_test[:1]) for clf in voting_clf.estimators_])
print(voting_clf.score(X_test, y_test))
```

C.R. 5 text
```
[1]
[array([1]), array([1]), array([0])]
0.912
```

And we can look at the performance of the voting classifier on the test set which is `0.912`, which shows, the voting classifier outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities,[5] then we can tell `sklearn` to predict the class with the highest class probability, averaged over all the individual classifiers.

[5] i.e., if they all have a `predict_proba()` method.

This is called **soft voting**, which often achieves higher performance than hard voting as it gives more weight to highly confident votes. All we need to do is set the voting classifier's voting hyperparameter to `soft`, and ensure that all classifiers can estimate class probabilities.

This is **NOT** the case for the SVC class by default, so you need to set its probability hyperparameter to `True`

> This will make the SVC class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method.

Let's try that:

C.R. 6

```python
voting_clf.voting = "soft"
voting_clf.named_estimators["svc"].probability = True
voting_clf.fit(X_train, y_train)
print(voting_clf.score(X_test, y_test))
```

C.R. 7

```text
0.92
```

We reach 92% accuracy simply by using soft voting.

To give a brief summary of what we have learned till now:

Hard Voting
: Takes a simple majority vote to decide the final prediction, based on the most frequent class predicted by individual models.

Soft Voting
: Considers the probability scores of each class predicted by individual models and averages them to produce a more refined final prediction.

  When dealing with imbalanced datasets, soft voting can help mitigate the bias towards the majority class by taking into account the probabilities of all classes.

3.2 Bagging and Pasting

Now we got a general idea of RF, let's look at methods to improve it's performance. A way to get a diverse set of classifiers is to use diverse training algorithms. An alternative approach is to use the same training algorithm for every predictor but train them on different random subsets of the training set.

- When sampling is performed with replacement, this method is called **bagging**.[6]
- When sampling is performed without replacement, it is called **pasting**.

[6] This is short for bootstrap aggregating.

> Both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor.

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the statistical mode for classification,[7] or the average for regression. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.

[7] i.e., the most frequent prediction, just like with a hard voting classifier.

Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

> Predictors can all be trained in parallel, via different Central Processing Unit (CPU) cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods as they scale very well.

3.2.1 Implementation

`sklearn` offers simple classes for both bagging and pasting:

> the `BaggingClassifier` class, or `BaggingRegressor` for regression.

The code below trains an ensemble of 500 decision tree classifiers where each is trained on 100 training instances **randomly sampled** from the training set with replacement.[8] The `n_jobs` parameter tells `sklearn` the number of CPU cores to use for training and predictions, and `-1` tells `sklearn` to use all available cores:

[8] This is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`.

C.R. 8
python
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
```

```
bag_clf.fit(X_train, y_train)
```
C.R. 9
python

A `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities,[9] which is the case with decision tree classifiers.

[9] i.e., if it has a `predict_proba()` method

Please have a look at **Fig.** 3.2, which compares the decision boundary of a single decision tree with the decision boundary of a bagging ensemble of 500 trees, both trained on the moons dataset.

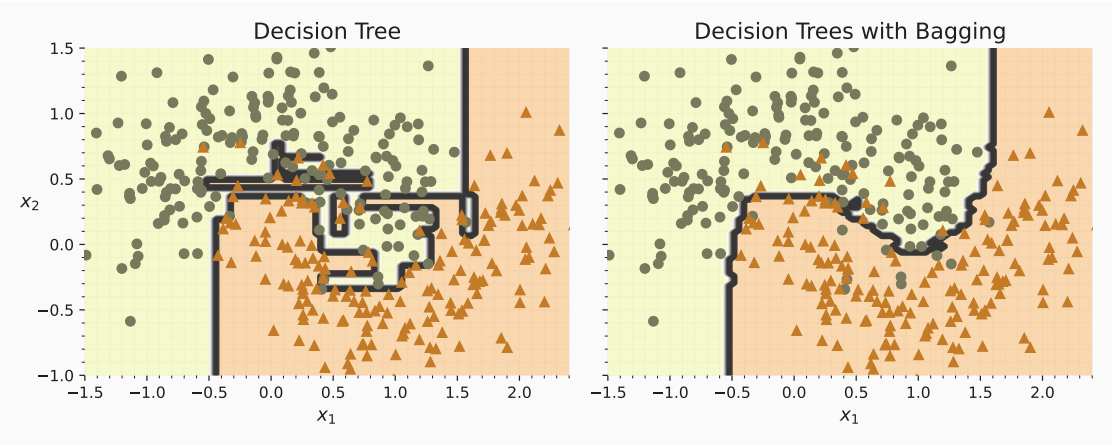


Figure 3.2: A single decision tree (left) versus a bagging ensemble of 500 trees (right)

As can be seen, the ensemble's predictions will likely generalize much better than the single decision tree's predictions:

the ensemble has a comparable bias but a smaller variance.

It makes roughly the same number of errors on the training set, but the decision boundary is less irregular.

Bagging introduces a higher diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced.

Overall, bagging often results in better models, which explains why it's generally preferred. However if we have spare time and CPU power, we can also use cross-validation to evaluate both bagging and pasting and select the one that works best.

3.2.2 Out-of-Bag Evaluation

With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples m training instances with replacement (`bootstrap=True`), where m is the size of the training set. With this process, it

can be shown mathematically that only about 63% of the training instances are sampled on average for each predictor.

Information: Limits of Bagging

If we were to randomly draw one instance from a dataset of size m, each instance in the dataset obviously has probability $\frac{1}{m}$ of getting picked, and therefore it has a probability $1-\frac{1}{m}$ of **NOT** getting picked.

If you draw m instances with **replacement**, all draws are independent and therefore each instance has a probability

$$\left(1-\frac{1}{m}\right)^m$$

of **NOT** getting picked. Now let's use the fact that $\exp x$ is equal to the limit of:

$$\exp x = \lim_{x\to\infty}\left(1+\frac{x}{m}\right)^m$$

So if we to assume m to be sufficiently large, the ratio of Out-of Bag (OOB) instances will be about $\exp -1 \simeq 0.37$. Therefore roughly 63% will be sampled ■

The remaining 37% of the training instances that are **NOT** sampled are called OOB instances.

> Note that they are **NOT** the same 37% for all predictors.

A bagging ensemble can be evaluated using OOB instances, without the need for a separate validation set as, if there are enough estimators, then each instance in the training set will likely be an OOB instance of several estimators, so these estimators can be used to make a fair ensemble prediction for that instance.

Once we have a prediction for each instance, we can determine the ensemble's prediction accuracy[10] In `sklearn`, we can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic OOB evaluation after training.

[10] Or any other metric of interest.

The following code demonstrates this effect:

C.R. 10

```python
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            oob_score=True, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
print(bag_clf.oob_score_)
```

The resulting evaluation score is available in the `oob_score_` attribute:

C.R. 11

```text
0.896
```

According to this OOB evaluation, this `BaggingClassifier` is likely to achieve about 89.6% accuracy on the test set. Let's verify this:

C.R. 12
```python
from sklearn.metrics import accuracy_score

y_pred = bag_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

C.R. 13
```text
0.912
```

We get 92% accuracy on the test. The OOB evaluation was a bit too pessimistic, just over 2% too low. The OOB decision function for each training instance is also available as the `oob_decision_function_` attribute. As the base estimator has a `predict_proba()` method, the decision function returns the class probabilities for each training instance.

For example, the OOB evaluation estimates that the first training instance has a 67.6% probability of belonging to the positive class and a 32.4% probability of belonging to the negative class:

C.R. 14
```python
print(bag_clf.oob_decision_function_[:3])  # probas for the first 3 instances
```

C.R. 15
```text
[[0.32352941 0.67647059]
 [0.3375     0.6625    ]
 [1.         0.        ]]
```

3.2.3 Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two (2) hyper-parameters:

- `max_features`,
- `bootstrap_features`.

They work the same way as `max_samples` and bootstrap, but for feature sampling instead of instance sampling. Therefore, each predictor will be trained on a **random subset of the input features**.

This technique is particularly useful when you are dealing with high-dimensional inputs,[11] as it can considerably speed up training.

[11] such as images.

> Sampling both training instances and features is called the random patches method.

Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than 1.0) is called the random subspaces method [13].

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

3.3 Random Forests

As we have discussed, a RF is an **ensemble of decision trees**, generally trained via the bagging method,[12] typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` class and passing it a `DecisionTreeClassifier`, we can just use the `RandomForestClassifier` class, which is more convenient and optimised for decision trees.[13] The following code trains a RF classifier with 500 trees, each limited to maximum 16 leaf nodes, using all available CPU cores:

[12] or sometimes pasting.

[13] similarly, there is a `RandomForestRegressor` class for regression tasks

C.R. 16

```python
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

With a few slight exceptions, a `RandomForestClassifier` has all the hyperparameters of the `DecisionTreeClassifier` class, which allows to control how trees are grown, in addition to all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.

The RF algorithm introduces **extra randomness** when growing trees. Instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features.

By default, it samples n features, where n is the total number of features. The algorithm results in greater tree diversity, which trades a higher bias for a lower variance, generally giving an overall better model.

So, the following `BaggingClassifier` is equivalent to the previous `RandomForestClassifier`:

C.R. 17

```python
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500, n_jobs=-1, random_state=42)
```

3.3.1 Extra-Trees

When we are growing a tree in a RF, at each node, only a random subset of the features is considered for splitting.[14] It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds.[15] For this, we can simply set `splitter="random"` when creating a `DecisionTreeClassifier`.

[14] as discussed previously.

[15] Which is what regular decision trees implement.

A forest of such extremely random trees is called an **extremely randomised trees**, or **ExtraTrees**. Here, values are chosen from a uniform distribution within the feature's empirical range.[16] Then, of all the randomly chosen splits, the one which produces the highest score is used to split the

[16] in the tree's training set.

node [15].

> As with previous methods, this technique trades more bias for a lower variance.

It also makes extra-trees classifiers much faster to train than regular RFs, as finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree [15].

We can create an extra-trees classifier using `sklearn`'s `ExtraTreesClassifier` class. Its Application-Programming Interface (API) is identical to the `RandomForestClassifier` class, except bootstrap defaults to `False`. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class, except bootstrap defaults to `False`.

> It is hard to tell in advance whether a `RandomForestClassifier` will perform better or worse than an `ExtraTreesClassifier`. Generally, the only way to know is to try both and compare them using cross-validation.

3.3.2 Feature Importance

Another great quality of RFs is that they make it easy to measure the relative importance of each feature. `sklearn` measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average, across all trees in the forest. More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it.

`sklearn` computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1. You can access the result using the `feature_importances_` variable. The following code trains a `RandomForestClassifier` on the *iris dataset* and outputs each feature's importance.

C.R. 18 python

```python
from sklearn.datasets import load_iris

iris = load_iris(as_frame=True)
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris.data, iris.target)
for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
    print(round(score, 2), name)
```

C.R. 19 text

```
0.11 sepal length (cm)
0.02 sepal width (cm)
0.44 petal length (cm)
0.42 petal width (cm)
```

Based on the results, it seems the most important features are the petal length (44%) and width

(42%), while sepal length and width are rather unimportant in comparison which are 11% and 2%, respectively.

Similarly, if we were to train a RF classifier on the Modified National Institute of Standards and Technology (MNIST) dataset and plot each pixel's importance, we get the image represented in **Fig.** 3.3.

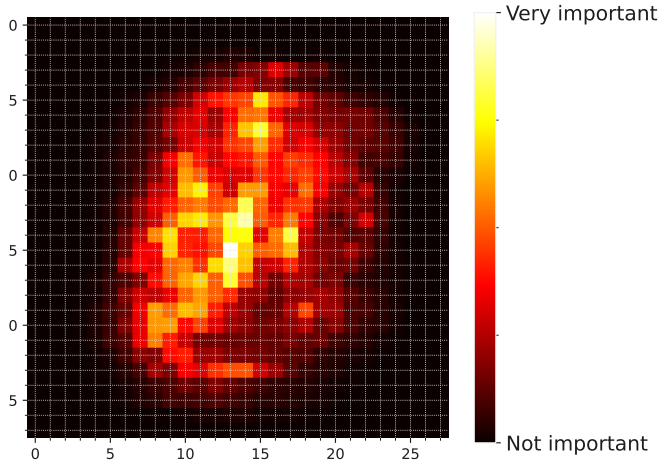


Figure 3.3: MNIST pixel importance (according to a RF classifier)

RFs are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

3.4 Boosting

Boosting refers to any ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to **train predictors sequentially**, each trying to correct its predecessor. The general structure of it is as follows:

- Initially, a model is built using the training data.
- Subsequent models are then trained to address the mistakes of their predecessors.
- Boosting assigns weights to the data points in the original dataset.

  Higher weights Instances which were misclassified by the previous model receive higher weights.

  Lower weights Instances which were correctly classified receive lower weights.
- Training on weighted data: The subsequent model learns from the weighted dataset, focusing its attention on harder-to-learn examples (those with higher weights).
- This iterative process continues until the entire training dataset is accurately predicted, or a predefined maximum number of models is reached.

There are many boosting methods available, but by far the most popular are **AdaBoost** and **gradient boosting**. Let's start with AdaBoost.[17]

[17] which is short for adaptive boosting.

3.4.1 AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances which the predecessor **underfit**. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, when training an AdaBoost classifier, the algorithm first trains a base classifier[18] and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Following this, it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on.

[18] i.e., a decision tree.

Fig. 3.4 shows the decision boundaries of five (5) consecutive predictors on the moons dataset.[19] The first classifier gets many instances wrong, so their weights get boosted.

[19] In this example, each predictor is a highly regularized SVM classifier with an RBF kernel.

The second classifier therefore does a better job on these instances, and so on. The plot on the right in **Fig.** 3.4 represents the same sequence of predictors, except that the learning rate is halved.[20] As can be seen, this sequential learning technique has some similarities with gradient descent, except instead of tweaking a single predictor's parameters to minimise a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

[20] i.e., the misclassified instance weights are boosted much less at every iteration.

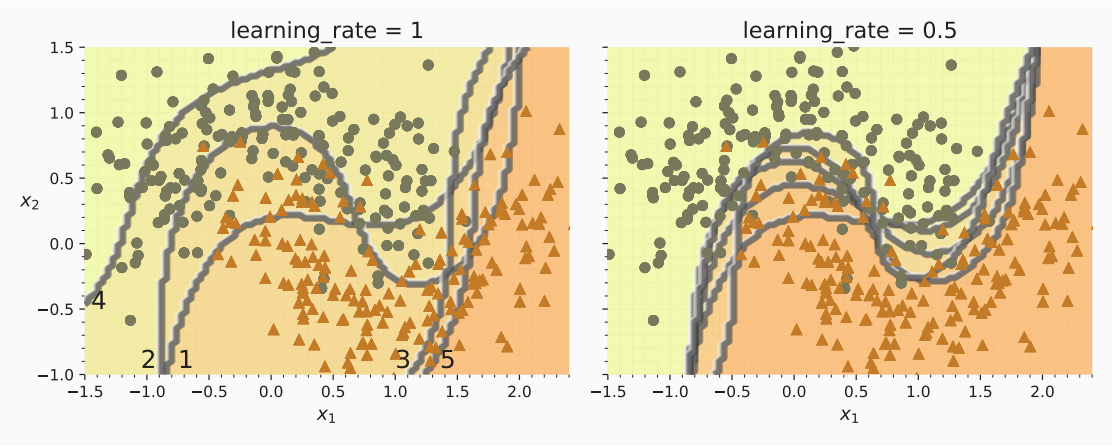


Figure 3.4: Decision boundaries of consecutive predictors.

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

> There is one important drawback to this sequential learning technique. Training cannot be parallelized as each predictor can only be trained after the previous predictor has been trained and evaluated. Therefore, it does **NOT** scale as well as bagging or pasting.

Let's take a closer look at the AdaBoost algorithm.

Each instance weight $w^{(i)}$ is initially set to $1/m$. A first predictor is trained, and its weighted error rate r_1 is computed on the training set.

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}} \tag{3.1}$$

where $\hat{y}_j^{(i)}$ is the j^{th} predictor's prediction for the i^{th} instance. The predictor's weight (α_j) is then determined using Eq. (3.2), where η is the learning rate hyperparameter.[21]

[21] which is 1 by default.

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j} \tag{3.2}$$

The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong,[22] then its weight will be negative.

[22] i.e., less accurate than random guessing.

Next, the AdaBoost algorithm updates the instance weights, using Eq. (3.3), which boosts the weights of the misclassified instances.

$$w^{(i)} = \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)}, \\ w^{(i)} \exp \alpha_j & \text{if } \hat{y}_j^{(i)} \neq y^{(i)}. \end{cases} \tag{3.3}$$

Then all the instance weights are normalised by dividing it with $\sum_{i=1}^{m} w^{(i)}$.

Finally, a new predictor is trained using the updated weights, and the whole process is repeated:

> the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on.

The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes.

`sklearn` uses a multi-class version of AdaBoost called Stagewise Additive Modeling using a Multiclass Exponential loss function (SAMME). When there are just two (2) classes, SAMME is equivalent to AdaBoost. If the predictors can estimate class probabilities,[23] `sklearn` can use a variant of SAMME called SAMME.R (the R stands for "Real"), which relies on class probabilities rather than predictions and generally performs better.

[23] i.e., if they have a `predict_proba()` method.

The following code trains an AdaBoost classifier based on 30 decision stumps using `sklearn`'s AdaBoostClassifier class.[24] A decision stump is a decision tree with `max_depth=1`, which in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

[24] As you might expect, there is also an AdaBoostRegressor class.

C.R. 20

```python
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=30,
    learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

If AdaBoost ensemble is overfitting the training set, we can try reducing the number of estimators or more strongly regularizing the base estimator.

3.4.2 Gradient Boosting

Another very popular boosting algorithm is gradient boosting. Just like AdaBoost, gradient boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However,

Advantages	Disadvantages
Can effectively combine multiple weak classifiers to create a strong classifier with high accuracy	Can be sensitive to outliers and noisy data
Can handle complex datasets and capture intricate patters by iteratively adapting to difficult examples	Training process can be computationally expensive, especially dealing with large datasets.
By focusing on misclassified examples and adjusting sample weights, AdaBoost mitigates the risk of overfitting	Appropriate selection of weak classifiers and the number of hyperparameters are crucial for performance.
A versatile algorithm which can work with different types of base classifiers	Can struggle with imbalanced datasets.

Table 3.1: The advantages and disadvantages of AdaBoost.

instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor.

Let's go through a simple regression example, using decision trees as the base predictors. This is called gradient tree boosting, or gradient boosted regression trees (GBRT). First, let's generate a noisy quadratic dataset and fit a `DecisionTreeRegressor` to it:

C.R. 21
```python
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100)  # y = 3xš + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

C.R. 22
```python
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
```

And then we'll train a third regressor on the residual errors made by the second predictor:

C.R. 23
```python
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three (3) trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

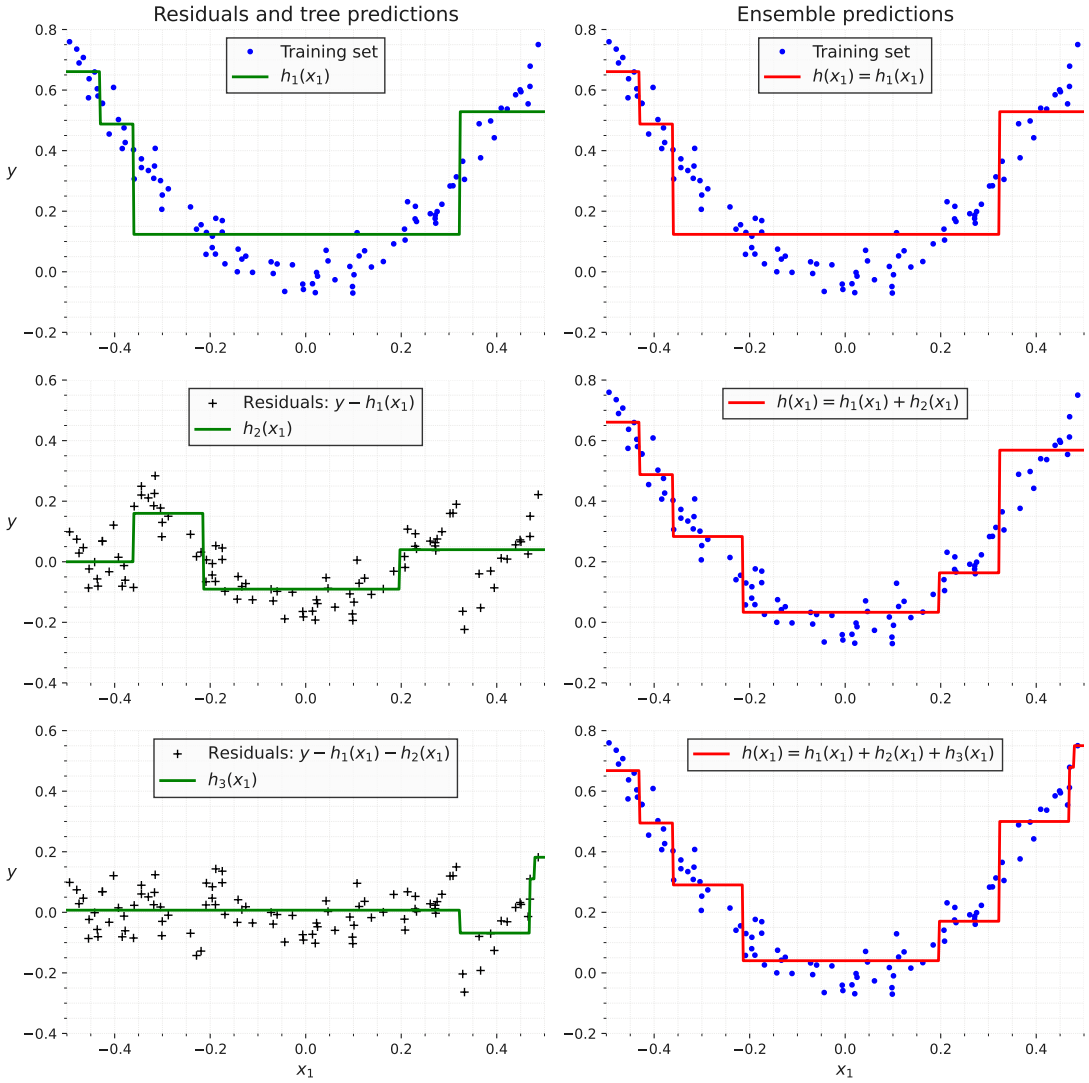


Figure 3.5: In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

C.R. 24
```python
X_new = np.array([[-0.4], [0.], [0.5]])
print(sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3)))
```

C.R. 25
```text
[0.49484029 0.04021166 0.75026781]
```

Fig. 3.5 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right we can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained

on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

We can use `sklearn`'s `GradientBoostingRegressor` class to train GBRT ensembles more easily. As a reminder, there's also a `GradientBoostingClassifier` class for classification. Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of decision trees, such as `max_depth`, `min_samples_leaf`, as well as hyperparameters to control the ensemble training, such as the number of trees.[25]

[25] i.e., `n_estimators`.

The following code creates the same ensemble as the previous one:

C.R. 26

```python
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                 learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
```

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.05, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called **shrinkage** where **Fig.** 3.6 shows two GBRT ensembles trained with different hyperparameters: the one on the left does not have enough trees to fit the training set, while the one on the right has about the right amount. If we added more trees, the GBRT would start to overfit the training set.

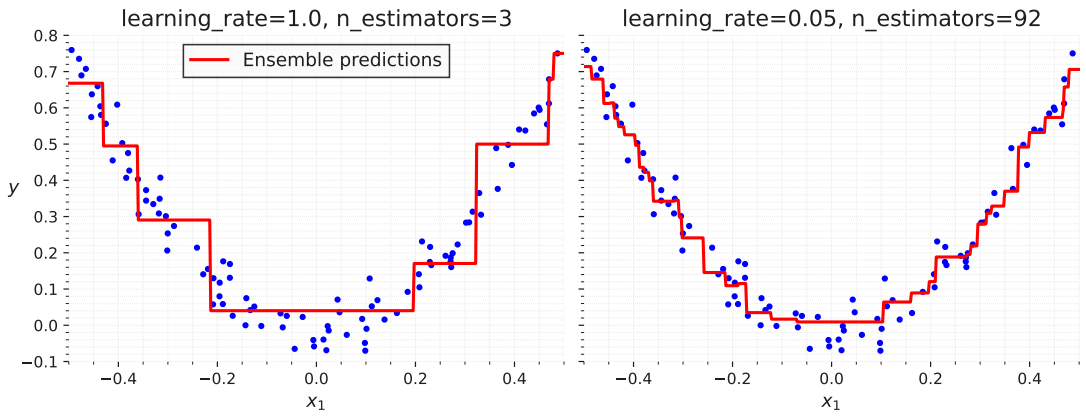


Figure 3.6: GBRT ensembles with not enough predictors (left) and just enough (right).

To find the optimal number of trees, we could perform cross-validation using `GridSearchCV` or `RandomizedSearchCV`, as usual, but there's a simpler way:

> if we set the `n_iter_no_change` hyperparameter to an integer value, say 10, then the `GradientBoostingRegressor` will automatically stop adding more trees during training if it sees that the last 10 trees didn't help.

This is simply early stopping, but with a little bit of patience. It tolerates having no progress for a

few iterations before it stops. Let's train the ensemble using early stopping:

C.R. 27
```python
gbrt_best = GradientBoostingRegressor(
    max_depth=2, learning_rate=0.05, n_estimators=500,
    n_iter_no_change=10, random_state=42)
gbrt_best.fit(X, y)

```

If you set `n_iter_no_change` too low, training may stop too early and the model will underfit. But if you set it too high, it will overfit instead. We also set a fairly small learning rate and a high number of estimators, but the actual number of estimators in the trained ensemble is much lower, thanks to early stopping:

C.R. 28
```python
print(gbrt_best.n_estimators_)
```

C.R. 29
```text
92
```

When `n_iter_no_change` is set, the `fit()` method automatically splits the training set into a smaller training set and a validation set: this allows it to evaluate the model's performance each time it adds a new tree. The size of the validation set is controlled by the `validation_fraction` hyperparameter, which is 10% by default. The `tol` hyperparameter determines the maximum performance improvement that still counts as negligible. It defaults to 0.0001. The `GradientBoostingRegressor` class also supports a subsample hyperparameter, which specifies the fraction of training instances to be used for training each tree.

For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called stochastic gradient boosting.

3.4.3 Histogram-Based Gradient Boosting

`sklearn` provides another GBRT implementation, optimised for large datasets: histogram based gradient boosting (HGB). It works by binning the input features, replacing them with integers. The number of bins is controlled by the `max_bins` hyperparameter, which defaults to 255 and cannot be set any higher than this. Binning can greatly reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more memory efficient data structures. And the way the bins are built removes the need for sorting the features when training each tree.

As a result, this implementation has a computational complexity of $\mathcal{O}(b \times m)$ instead of $\mathcal{O}(n \times m \log m)$, where b is the number of bins, m is the number of training instances, and n is the number of features. In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets. However, binning causes a precision loss, which acts as a regularizer: depending on the

dataset, this may help reduce overfitting, or it may cause underfitting.

`sklearn` provides two (2) classes for HGB:

1. `HistGradientBoostingRegressor`
2. `HistGradientBoostingClassifier`

They're similar to `GradientBoostingRegressor` and `GradientBoostingClassifier`, with a few notable differences:

- Early stopping is automatically activated if the number of instances is greater than 10,000. You can turn early stopping always on or always off by setting the `early_stopping` hyperparameter to True or False.
- Subsampling is not supported.
- `n_estimators` is renamed to `max_iter`.
- The only decision tree hyperparameters that can be tweaked are:
  - `max_leaf_nodes`,
  - `min_samples_leaf`,
  - and `max_depth`.

The HGB classes also have two (2) nice features:

> They support both categorical features and missing values. This simplifies pre-processing quite a bit.

However, the categorical features must be represented as integers ranging from 0 to a number lower than `max_bins`. You can use an `OrdinalEncoder` for this.

For example, here's how to build and train a complete pipeline for the California housing dataset:

C.R. 30

python

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder
hgb_reg = make_pipeline(
make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
remainder="passthrough"),
HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)
```

The whole pipeline is just as short as the imports! No need for an imputer, scaler, or a one-hot

encoder, so it's really convenient. Note that `categorical_features` must be set to the categorical column indices (or a Boolean array). Without any hyperparameter tuning, this model yields an RMSE of about 47,600, which is not too bad.

Information: Implementations

Several other optimised implementations of gradient boosting are available in the Python ML ecosystem: in particular, XGBoost, CatBoost, and LightGBM. These libraries have been around for several years. They are all specialized for gradient boosting, their APIs are very similar to `sklearn`'s, and they provide many additional features, including GPU acceleration; you should definitely check them out! Moreover, the TensorFlow RFs library provides optimised implementations of a variety of RF algorithms, including plain RFs, extra-trees, GBRT, and several more.

3.5 Bagging v. Boosting

Similarities

Bagging and Boosting, both being the commonly used methods, have a universal similarity of being classified as ensemble methods. To summarise, let's look at the similarities between them.

- Both are ensemble methods to get N learners from 1 learner.
- Both generate several training data sets by random sampling.
- Both make the final decision by averaging the N learners (or taking the majority of them i.e Majority Voting).
- Both are good at reducing variance and provide higher stability.

Differences

Bagging	Boosting
The simplest way of combining predictions that belong to the same type. classifiers to create a strong classifier with high accuracy	A way of combining predictions that belong to the different types.
Aim to decrease variance, not bias	Aim to decrease bias, not variance.
Each model receives equal weight.	Models are weighted according to their performance.
Each model is built independently.	New models are influenced by the performance of previously built models.
Different training data subsets are selected using row sampling with replacement and random sampling methods from the entire training dataset.	Iteratively train models, with each new model focusing on correcting the errors (misclassifications or high residuals) of the previous models
Bagging tries to solve the over-fitting problem.	Boosting tries to reduce bias.
In this base classifiers are trained in parallel.	In this base classifiers are trained sequentially.

Table 3.2: A comparison between Bagging and Boosting.

3.6 Stacking

The last ensemble method we will discuss, is called stacking.[26] It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?

[26] short for stacked generalization.

To train the blender, you first need to build the blending training set. You can use `cross_val_predict()` on every predictor in the ensemble to get out-of-sample predictions for each instance in the original training set, and use these can be used as the input features to train the blender; and the targets can simply be copied from the original training set. Note that regardless of the number of features in the original training set (just one in this example), the blending training set will contain one input feature per predictor (three in this example). Once the blender is trained, the base predictors are retrained one last time on the full original training set.

It is actually possible to train several different blenders this way (e.g., one using linear regression, another using RF regression) to get a whole layer of blenders, and then add another blender on top of that to produce the final prediction. You may be able to squeeze out a few more drops of performance by doing this, but it will cost you in both training time and system complexity.

`sklearn` provides two classes for stacking ensembles: StackingClassifier and StackingRegressor. For example, we can replace the VotingClassifier we used at the beginning of this chapter on the moons dataset with a `StackingClassifier`:

C.R. 31

python

```
from sklearn.ensemble import StackingClassifier
stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),

cv=5 # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

For each predictor, the stacking classifier will call `predict_proba()` if available; if not it will fall back to `decision_function()` or, as a last resort, call `predict()`. If you don't provide a final estimator, StackingClassifier will use `LogisticRegression` and StackingRegressor will use RidgeCV. If you evaluate this stacking model on the test set, you will find 92.8% accuracy, which is a bit better than the voting classifier using soft voting, which got 92%.

In conclusion, ensemble methods are versatile, powerful, and fairly simple to use. RFs, AdaBoost, and GBRT are among the first models you should test for most ML tasks, and they particularly shine with heterogeneous tabular data. Moreover, as they require very little preprocessing, they're great for getting a prototype up and running quickly. Lastly, ensemble methods like voting classifiers

and stacking classifiers can help push your system's performance to its limits.

Chapter 4

Dimensionality Reduction

Table of Contents

4.1 Introduction

Many ML problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, which we will see in the continuing parts of this chapter. This problem is often referred to as the curse of dimensionality.[1]

[1] Refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience [16].

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one.

For example, consider the MNIST images we worked previously. The pixels on the image borders are almost always white, so we could completely drop these pixels from the training set without losing any information.

As we saw previously in Ensemble Learning, we have confirmed these pixels are unimportant for the classification task, shown in **Fig.** 3.3. Additionally, two (2) neighboring pixels are often **highly correlated**:

if we merge them into a single pixel (e.g., by taking the mean of the two pixel intensities),

we will not lose much information.

Information: Limits of Reduction

Reducing dimensionality does cause some **information loss**, similar to compressing an image to JPEG can degrade its quality, so even though it will speed up training, it may make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain.

Therefore, it is in our best interest to first try to train your system with the **original data** before considering using dimensionality reduction.

In some cases, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance, but in general it won't. It will just speed up training.

Apart from speeding up training, dimensionality reduction is also highly useful for data visualisation. Reducing the number of dimensions down to two (2), or three (3), makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters.[2] Moreover, data visualisation is essential to communicate our conclusions to people who are not data scientists-in particular, decision makers who will use your results.

[2] A type of plot or mathematical diagram using Cartesian coordinates to display values for typically two variables for a set of data.

In this chapter, we will first discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Following this, we will consider the two (2) main approaches to dimensionality reduction:

- projection, and
- manifold learning,

and we will go through three (3) of the most popular dimensionality reduction techniques:

1. PCA,
2. Random projection,
3. Locally Linear Embedding (LLE).

4.1.1 The Problems of Dimensions

We are used to living in three dimensions that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our minds, let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

> The more dimensions the data has, the less geometrically explainable it becomes.

Turns out many things behave very differently in high-dimensional space. As an example, picking

a random point in a unit square (a 1-by-1 square), will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be "extreme" along any dimension). However, if we do the same thin in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border which as can be seen is very counter-intuitive.

Theory 4.2: Distance Between Points in n-Dimensions

Let d be the dimension of a cube in d-dimensions. A point within the cube will have 2 boundaries. To be less than 0.001 from a boundary in a d-dimensional unit cube, means it is not inside the cube of side length $1 - 2 \times 0.001$ sharing the same origin point as the original cube.

$$\begin{aligned} \text{2D} \quad & \left(1 - (1 - 2 \times 0.001)^2\right) \times 100\% \sim 0.3996\% \\ \text{10,000 D} \quad & \left(1 - (1 - 2 \times 0.001)^{10,000}\right) \times 100\% \sim 99.999999798\% \end{aligned}$$ ■

Information: n-dimensional Geometry

When dimensions become detached from real-life, the mathematics and our intuition becomes more divergent. There are many problems in mathematics where as the geometry becomes n-dimensional the results get even more complicated.

Here is another example.

If we were to pick two (2) points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If we were to pick two random points in a 3D unit cube instead, the average distance will be roughly 0.66.

But what about two points picked randomly in a 1,000,000-dimensional unit hypercube?

The average distance, believe it or not, will be about 408.25. This is a problem known as **Hypercube Line Picking**

Theory 4.3: Hypercube Line Picking

Let two points x and y be picked randomly from a unit n-dimensional hypercube. The expected distance between the points $\Delta(n)$, i.e., the mean line segment length, is then:

$$\Delta(n) = \int_0^1 \cdots \int_0^1 \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 \cdots (x_n - y_n)^2}\, dx_1 \cdots dx_n\, dy_1 \cdots dy_n$$

The first few values for $\Delta(n)$ are given in the following table.

n	OEIS	$\Delta(n)$	n	OEIS	$\Delta(n)$
1	–	0.3333333333...	5	A103984	0.8785309152...
2	A091505	0.5214054331...	6	A103985	0.9689420830...
3	A073012	0.6617071822...	7	A103986	1.0515838734...
4	A103983	0.7776656535...	8	A103987	1.1281653402...

This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, **there's just plenty of space in high dimensions**. As a result, high-dimensional datasets are at risk of being very sparse:

> most training instances are likely to be far away from each other.

This also means a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations.

> The more dimensions the training set has, the greater the risk of overfitting it.

In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions.

With just 100 features, which is significantly fewer than in the MNIST problem, all ranging from 0 to 1, you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

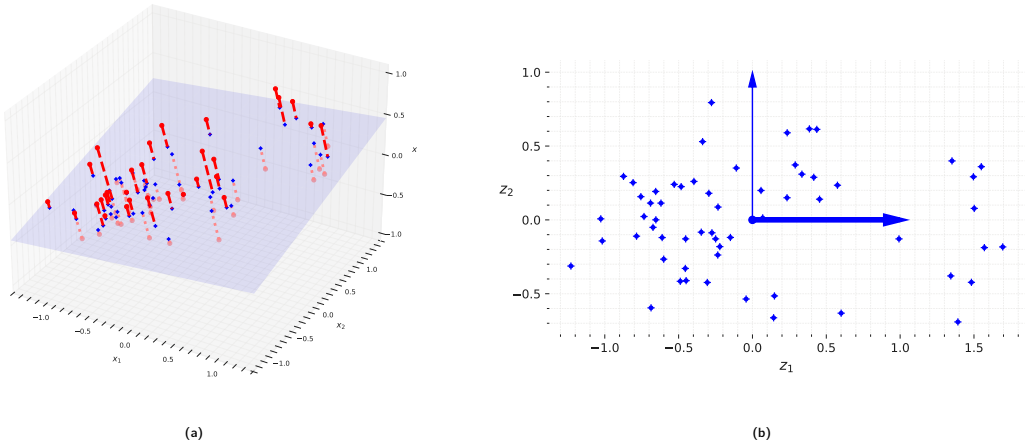


Figure 4.1: (a) A 3D dataset lying close to a 2D subspace (b) The new 2D dataset after projection

4.2 Main Approaches to Dimensionality Reduction

Before we dive into different dimensionality reduction algorithms, let's take a look at the two (2) main approaches to reducing dimensionality:

1. projection, and
2. manifold learning.

4.2.1 Projection

In most real-world problems, training instances are **NOT** spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, all training instances lie within a much lower-dimensional subspace of the high-dimensional space. This sounds very abstract, so let's look at an example.

In **Fig.** 4.1a we can see a 3D dataset represented by **small spheres**.

As you can see, almost all training instances lie close to a plane:

> this is a lower-dimensional (2D) subspace of the higher-dimensional (3D) space.

If we were to project every training instance perpendicularly onto this subspace,[3] we get the new 2D dataset shown in **Fig.** 4.1b.

[3] as represented by the short dashed lines connecting the instances to the plane

We have just reduced the dataset's dimensionality from 3D to 2D. Please observe the axes corresponding to new features z_1 and z_2 which are the coordinates of the projections on the plane.

4.2.2 Manifold Learning

It is worth mentioning that, projection is not always the best approach to dimensionality reduction. In many cases the subspace may **twist** and **turn**, such as in the famous Swiss roll toy dataset[4] represented in **Fig.** 4.2.

[4] The Swiss roll is a toy dataset in `sklearn` that is commonly used for testing and demonstrating nonlinear dimensionality reduction algorithms. It consists of a set of points in three dimensions, arranged in a roll shape, such that the points on the roll are mapped to a two-dimensional plane in a nonlinear fashion

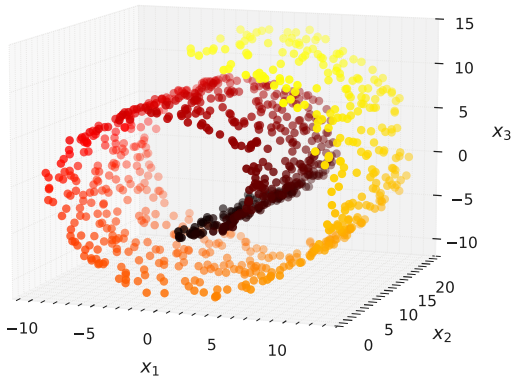
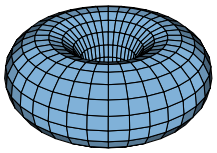


Figure 4.2: The Swiss roll dataset

Simply projecting onto a plane[5] would squash different layers of the Swiss roll together, as shown on the left side of **Fig.** 4.3. What we probably want instead is to unroll the Swiss roll to obtain the 2D dataset on the right side of **Fig.** 4.3.

[5] e.g., by dropping the dimension x_3

The Swiss roll is an example of a 2D manifold. To put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space.[6] More generally, a d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane.



[6] A torus can be taught of as a 2D manifold as the entire surface is defined in a 2D space.

In the case of the Swiss roll, $d = 2$ and $n = 3$.

> It locally resembles a 2D plane, but it is rolled in the 3rd dimension.

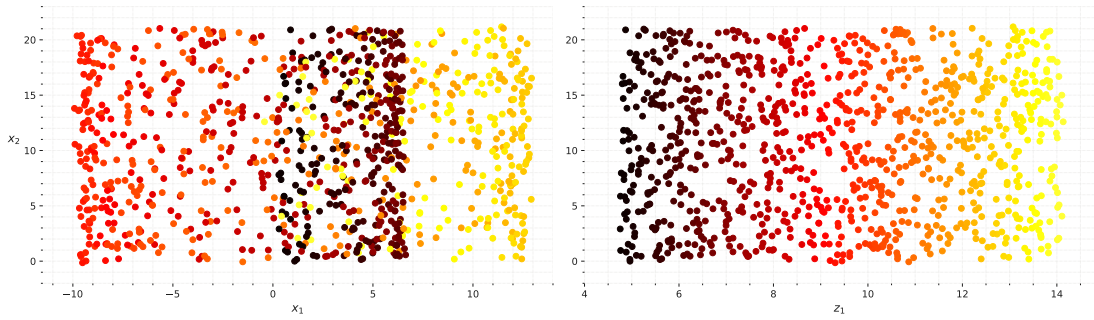


Figure 4.3: Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Many dimensionality reduction algorithms work by modelling the manifold on which the training instances lie. This is called manifold learning [17]. It relies on the manifold assumption, also called the *manifold hypothesis*:

> Most real-world high-dimensional datasets lie close to a much lower dimensional manifold.

This assumption is very often empirically[7] observed.

[7] This means it is based on, concerned with, or verifiable by observation or experience rather than theory or pure logic

Once again, let us look back at the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, and they are more or less centered. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits.

In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want.

> These constraints tend to squeeze the dataset into a lower-dimensional manifold.

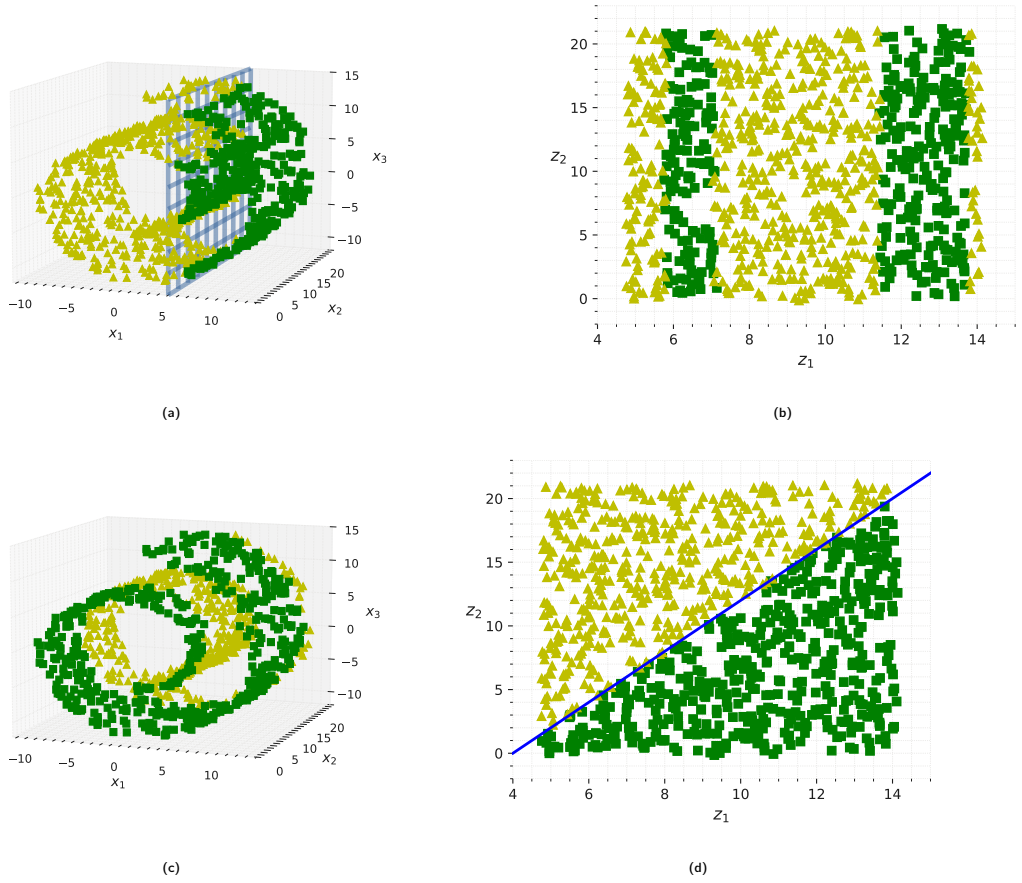


Figure 4.4: The decision boundary may not always be simpler with lower dimensions.

The manifold assumption is often accompanied by another **implicit** assumption:

> The task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold.

For example, in **Fig.** 4.4c, the Swiss roll is split into two (2) classes. In the 3D space **Fig.** 4.4a the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right) the decision boundary is a straight line.

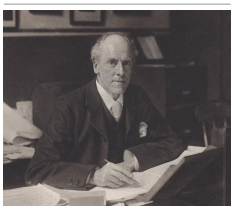
However, this implicit assumption does not always hold. For example, in the bottom row of Figure 8-6, the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

> Reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

We now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms for dimensionality reduction.

4.3 Principal Component Analysis (PCA)

PCA is by far the most popular dimensionality reduction algorithm, invented in 1901 by Karl Pearson.[8] First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it, just like in **Fig.** 4.1a.



[8] **Karl Pearson**
(1857 - 1936)

An English biostatistician and mathematician. He has been credited with establishing the discipline of mathematical statistics. He founded the world's first university statistics department at University College London in 1911, and contributed significantly to the field of biometrics and meteorology.

4.3.1 Preserving the Variance

Before we can project the training set onto a lower-dimensional hyperplane, we first need to choose the correct hyperplane. As an example, a simple 2D dataset is represented on the left in **Fig.** 4.5 along with three different axes, such as 1D hyperplanes.

On the right is the result of the projection of the dataset onto each of these axes. As we can see:

Top The projection onto the solid line preserves the maximum variance,

Bottom The projection onto the dotted line preserves very little variance

Middle The projection onto the dashed line preserves an intermediate amount of variance

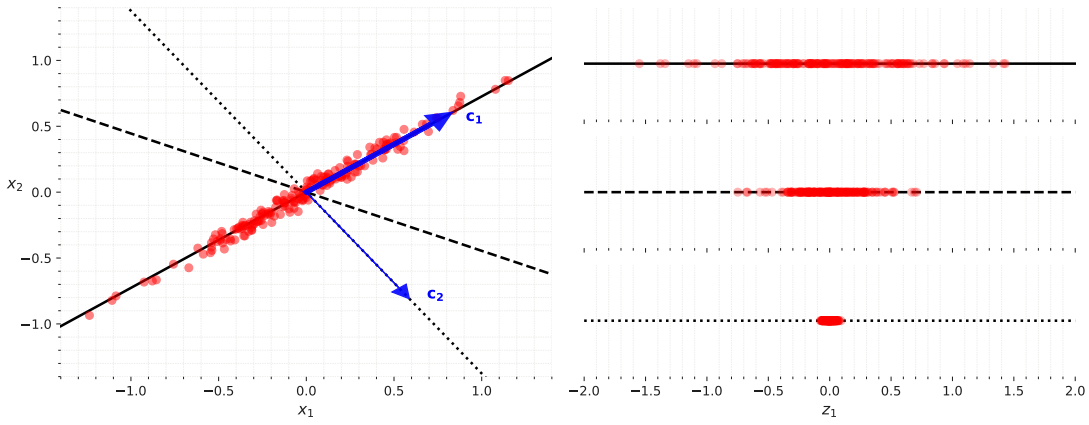


Figure 4.5: Selecting the subspace on which to project.

It seems reasonable to select the **axis preserving the maximum amount of variance**, as it will most likely lose less information than the other projections.

> Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis.

This is **core idea** behind PCA.

4.3.2 Principal Components

PCA identifies the axis accounting for the **largest amount of variance** in the training set. In **Fig.** 4.5, it is the solid line.

It also finds a second axis, **orthogonal** to the first one, which accounts for the largest amount of the remaining variance. In this 2D example there is no choice as it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a 3rd axis, orthogonal to both previous axes, and a fourth, a fifth, and so on, as many axes as the number of dimensions in the dataset.

The i^{th} axis is called the i^{th} Principal Component (PC) of the data. In **Fig.** 4.5, the first PC is the axis on which vector c_1 lies, and the second PC is the axis on which vector c_2 lies.

In **Fig.** 4.1a the first two PCs are on the projection plane, and the third PC is the axis orthogonal to that plane. After the projection, in **Fig.** 4.1b, the first PC corresponds to the z_1 axis, and the second PC corresponds to the z_2 axis.

Information: The Idea Behind PCA

For each PC, PCA finds a zero-centered unit vector pointing in the direction of the PC. As two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is not stable: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes. In some cases, a pair of unit vectors may even rotate or swap,[9] but the plane they define will generally remain the same.

[9] if the variances along these two axes are very close

So how can we find the principal components of a training set?

Luckily, there is a standard matrix factorisation technique called Singular Value Decomposition (SVD) which decomposes the training set matrix $\boldsymbol{X}$ into the matrix multiplication of three (3) matrices $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, where $\boldsymbol{V}$ contains the unit vectors that define all the principal components that you are looking for, as shown below:

$$\boldsymbol{V} = \begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix}$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the 3D training set represented in **Fig.** 4.1a, then it extracts the two unit vectors that define the first two PCs:

C.R. 1

```python
import numpy as np

X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

PCA assumes that the dataset is centered around the origin. As we will see, `sklearn`'s PCA classes take care of centering the data for us. If we were to implement PCA ourselves, or if we used other libraries, we shouldn't forget to center the data first.

4.3.3 Downgrading Dimensions

Once we identified all the PCs, we can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible.

For example, in **Fig.** 4.1a the 3D dataset is projected down to the 2D plane defined by the first two (2) principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane and obtain a reduced dataset $\boldsymbol{X}_{\text{d}-\text{proj}}$ of dimensionality d, compute the matrix multiplication of the training set matrix $\boldsymbol{X}$ by the matrix W_{d}, defined as the matrix containing the first d columns of $\boldsymbol{V}$:

$$\boldsymbol{X}_{\text{d}-\text{proj}} = \boldsymbol{X}\boldsymbol{W}_{\text{d}}$$

The following Python code projects the training set onto the plane defined by the first two principal components:

C.R. 2
```python
W2 = Vt[:2].T
X2D = X_centered @ W2
```

We now know how to reduce the dimensionality of any dataset by projecting it down to any number of dimensions, while preserving as much variance as possible.

Using sklearn The `PCA` class uses SVD to implement PCA, just like we did earlier. The following code applies PCA to reduce the dimensionality of the dataset down to two (2) dimensions:

`sklearn`'s `PCA` also automatically takes care of centering the data.

C.R. 3
```python
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

After fitting the PCA transformer to the dataset, its `components_` attribute holds the transpose of $\boldsymbol{W}_{\text{d}}$: it contains one row for each of the first d principal components.

Explained Variance Ratio

Another useful piece of information is the explained variance ratio of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component.

For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in **Fig.** 4.1a:

C.R. 4 python
```python
print(pca.explained_variance_ratio_)
```

C.R. 5 text
```
[0.17974135 0.1177597 ]
```

This output tells us that about 76% of the dataset's variance lies along the first PC, and about 15% lies along the second PC. This leaves about 9% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

4.3.4 The Right Number of Dimensions

Instead of randomly choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions which add up to the required summed variance, for example, 95%.

> An exception to this rule, of course, is if you are reducing dimensionality for data visualization, in which case you will want to reduce the dimensionality down to 2 or 3.

Let's load and splits the MNIST dataset and performs PCA without reducing dimensionality, Then compute the minimum number of dimensions required to preserve 95% of the training set's variance:

C.R. 6 python
```python
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False, parser="auto")
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1  # d equals 154
```

We could then set `n_components=d` and run PCA again, but there's a better option. Instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

C.R. 7 python
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

The actual number of components is determined during training, and it is stored in the `n_components_` attribute:

C.R. 8 python
```
print(pca.n_components_)
```

C.R. 9 text
```
154
```

A different option is to plot the explained variance as a function of the number of dimensions which you can see in **Fig.** 4.6. There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, we can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

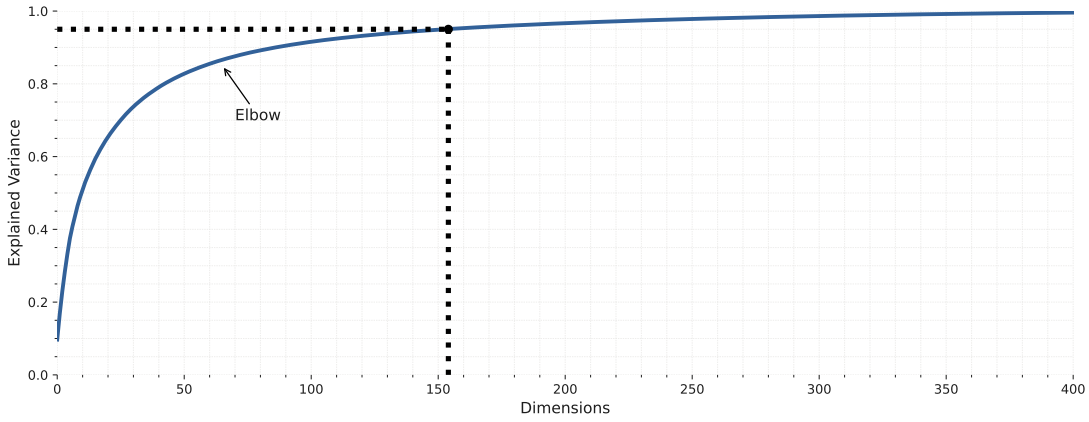


Figure 4.6: Explained variance as a function of the number of dimensions.

Finally, if we are using dimensionality reduction as a pre-processing step for a supervised learning task, then we can tune the number of dimensions as you would any other hyperparameter.

For example, the following code example creates a two-step pipeline, first reducing dimensionality using PCA, then classifying using a random forest. Next, it uses `RandomizedSearchCV` to find a good combination of hyperparameters for both PCA and the random forest classifier.

This example does a quick search, tuning only two (2) hyperparameters, training on just 1,000 instances, and running for just 10 iterations:

C.R. 10 python
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

clf = make_pipeline(PCA(random_state=42),
```

C.R. 11
```python
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])


```

Let's look at the best hyperparameters found:

C.R. 12
```python
print(rnd_search.best_params_)
```

C.R. 13
```text
{'randomforestclassifier__n_estimators': 475, 'pca__n_components': 57}
```

It's interesting to see how low the optimal number of components is:

> we reduced a 784-dimensional dataset to just 57 dimensions.

This is tied to the fact that we used a random forest, which is a pretty powerful model. If we used a linear model instead, such as an `SGDClassifier`, the search would find that we need to preserve more dimensions.

4.3.5 PCA for Compression

After dimensionality reduction, the training set takes up much less space. For example, after applying PCA to the MNIST dataset while preserving 95% of its variance, we are left with 154 features, instead of the original 784 features. So the dataset is now less than 20% of its original size, and we only lost 5% of its variance. This is a reasonable compression ratio, and it's easy to see how such a size reduction would speed up a classification algorithm tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the **reconstruction error** [18].

The `inverse_transform()` method lets us decompress the reduced MNIST dataset back to 784 dimensions:

C.R. 14
```python
X_recovered = pca.inverse_transform(X_reduced)
```

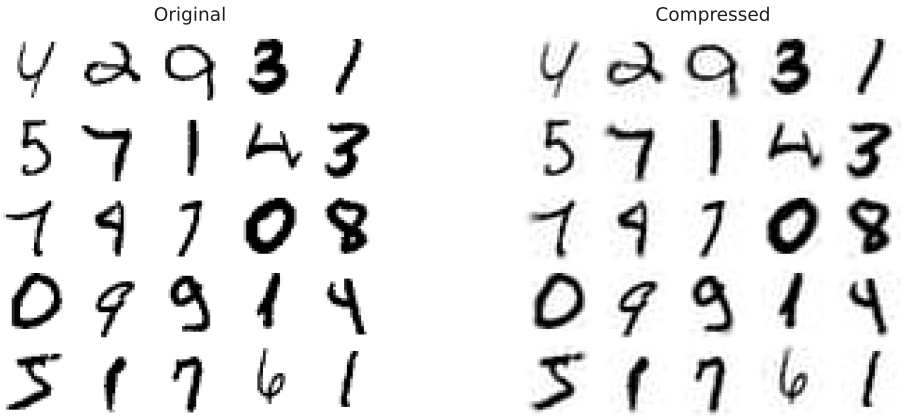


Figure 4.7: MNIST compression that preserves 95% of the variance

Fig. 4.7 shows a few digits from the original training set, seen on the left, and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact. The equation for the inverse transformation is shown below.

$$\boldsymbol{X}_{\text{recovered}} = \boldsymbol{X}_{\text{d-proj}} \boldsymbol{W}_{\text{d}}^{\top}$$

4.3.6 Randomized PCA

If you set the `svd_solver` hyperparameter to `randomized`, `sklearn` uses a stochastic algorithm called randomised PCA that quickly finds an approximation of the first d principal components. Its computational complexity is:

$$\mathcal{O}\left(m \times d^2\right) + \mathcal{O}\left(d^3\right) \qquad \text{Instead of} \qquad \mathcal{O}\left(m \times n^2\right) + \mathcal{O}\left(n^3\right) \tag{4.1}$$

for full SVD approach, therefore it is faster than full SVD when d is much smaller than n:

C.R. 15

```python
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

By default, `svd_solver` is set to `"auto"`, where `sklearn` automatically uses the randomised PCA algorithm if $max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $min(m, n)$, or else it uses the full SVD approach. So the preceding code would use the randomized PCA algorithm even if you removed the `svd_solver="randomized"` argument, as $154 < 0.8 \times 784$.

If we want to force `sklearn` to use full SVD for a slightly more precise result, you can set the `svd_solver` hyperparameter to `"full"`.

4.3.7 Incremental PCA

A problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, incremental PCA algorithms have been developed that allow you to split the training set into mini-batches and feed these in one mini-batch at a time. This is useful for large training sets and for applying PCA online.[10]

[10] i.e., on the fly, as new instances arrive.

The following splits the MNIST training set into 100 mini-batches[11] and feeds them to `sklearn`'s Incremental PCA class to reduce the dimensionality of the MNIST dataset down to 154 dimensions, just like before.

[11] using NumPy's `array_split()` function

> We must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set.

C.R. 16

```python
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternatively, we can use NumPy's `memmap` class[12], which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory.

[12] Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory

To demonstrate this, let's first create a memory-mapped (**memmap**) file and copy the MNIST training set to it, then call `flush()` to ensure that any data still in the cache gets saved to disk. In real life, `X_train` would typically not fit in memory, so you would load it chunk by chunk and save each chunk to the right part of the memmap array:

C.R. 17

```python
filename = "my_mnist.mmap"
X_mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X_mmap[:] = X_train  # could be a loop instead, saving the data chunk by chunk
X_mmap.flush()
```

Next, we can load the memmap file and use it like a regular NumPy array. Let's use the `IncrementalPCA` class to reduce its dimensionality. Since this algorithm uses only a small part of the array at any given time, memory usage remains under control. This makes it possible to call the usual `fit()` method instead of `partial_fit()`, which is quite convenient:

C.R. 18

```python
X_mmap = np.memmap(filename, dtype="float32", mode="readonly").reshape(-1, 784)
batch_size = X_mmap.shape[0] // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mmap)
```

Only the raw binary data is saved to disk, so specify the data type and shape of the array when you load it.

If you omit the shape, `np.memmap()` returns a 1D array.

For very high-dimensional datasets, PCA can be too slow. As we saw previously, even if you use randomized PCA its computational complexity is still

$$\mathcal{O}\left(m \times d^2\right) + \mathcal{O}\left(d^3\right),$$

so the target number of dimensions d cannot be too large. If we are dealing with a dataset with tens of thousands of features or more,[13] then training may become much too slow: in this case, you should consider using random projection instead.

[13] For example images.

4.4 Random Projection

As its name suggests, the random projection algorithm projects the data to a lower-dimensional space using a random linear projection. This may sound counter intuitive, but turns out such a random projection is actually very likely to preserve distances fairly well, as was demonstrated mathematically by William B. Johnson and Joram Lindenstrauss in a famous lemma shown in an abridged form below [19].

Theory 4.4: Johnson - Lindenstrauss Lemma

For $0 < \epsilon < 1$, let $V = \{x_i : i = 1, \dots M\} \in \mathbb{R}$ be a set of points in $\mathbb{R}^m$ where m is the number of dimensions of the original dataset. If we define a lower dimension such as:

$$n \geq \frac{c}{\epsilon^2} \log M$$

Then there exist a linear mapping of $\mathbb{R}^m \to \mathbb{R}^n$ such for all $i \neq j$:

$$1 - \epsilon \leq \frac{\|A(x_i) - A\left(x_j\right)\|}{\|x_i - x_j\|} \leq 1 + \epsilon$$

The Theorem states that after fixing an error level, one can map a collection of points from one Euclidean space, no matter how high it's dimension m is to a smaller Euclidean space while only changing the distance between any two points by a factor of $1 \pm \epsilon$. The dimension of the image space is only dependent on the error and the number of points. Given that the dimension is very large, one can achieve significant dimension reduction ■

So, two similar instances will remain similar after the projection, and two very different instances will remain very different.

As intuition goes, the more dimensions we drop, the more information is lost, and the more distances get distorted.

So how can we choose the optimal number of dimensions?

Well, the progenitors of the aforementioned lemma defined a method which determines the minimum number of dimensions to preserve in order to ensure-with high probability-that distances won't change by more than a given tolerance.

For example, if we have a dataset containing $m = 5,000$ instances with $n = 20,000$ features each, and don't want the squared distance between any two instances to change by more than $\epsilon = 10\%$, then we should project the data down to d dimensions, with:

$$d \geq 4 \frac{\log m}{1/2\epsilon^2 - 1/3\epsilon^3}$$

which is 7,300 dimensions, which is a significant dimensionality reduction. Please observe that the equation does not use n, it only relies on m and ϵ. This equation is implemented by the `johnson_lindenstrauss_min_dim()` function:

Now we can just generate a random matrix $\boldsymbol{P}$ of shape (d, n), where each item is sampled randomly

from a Gaussian distribution with ($\mu = 0$, $\sigma^2 = 1/d$), and use it to project a dataset from n dimensions down to d:

Simple and efficient, and no training is required. The only thing the algorithm needs to create the random matrix is the dataset's shape. The data itself is not used at all.

For implementation `sklearn` offers a `GaussianRandomProjection` class. When we call its `fit()` method, it uses `johnson_lindenstrauss_min_dim()` to determine the output dimensionality, then it generates a random matrix, which it stores in the `components_` attribute. Then when we call `transform()`, it uses this matrix to perform the projection. When creating the transformer, we can set `eps` if we want to tweak ϵ,[14] and `n_components` if we want to force a specific target dimensionality d.

[14] The default value is 0.1

The following code example gives the same result as the preceding code which we can also use to verify that `gaussian_rnd_proj.components_` is equal to $\boldsymbol{P}$:

`sklearn` also provides a second random projection transformer, known as `SparseRandomProjection`. It determines the target dimensionality in the same way, generates a random matrix of the same shape, and performs the projection identically. The main difference is that the random matrix is sparse. This means it uses much less memory: about 25 MB instead of almost 1.2 GB in the preceding example! And it's also much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster in this case. Moreover, if the input is sparse, the transformation keeps it sparse (unless you set `dense_output=True`).

Finally, it enjoys the same distance-preserving property as the previous approach, and the quality of the dimensionality reduction is comparable. In short, it's usually preferable to use this transformer instead of the first one, especially for large or sparse datasets.

The ratio (r) of nonzero items in the sparse random matrix is called its **density**. By default, it is equal to $1/n$. With 20,000 features, this means that roughly only $1/141$ cells in the random matrix is non-zero.[15] We can set this density hyperparameter to another value if we prefer.

[15] There is a reason why it is called a sparse matrix.

Each cell in the sparse random matrix has a probability r of being non-zero, and each non-zero value is either $-v$ or $+v$,[16] where $v = 1/dr$.

[16] Which are both equally likely.

If we want to perform the inverse transform, you first need to compute the pseudo-inverse of the components matrix using SciPy's `pinv()` function, then multiply the reduced data by the transpose of the pseudo-inverse:

Random projection is a simple, fast, memory-efficient, and powerful dimensionality reduction algorithm that we should keep in mind, especially when dealing with high-dimensional datasets.

4.5 Locally Linear Embedding

Locally linear embedding(LLE) is a non-linear dimensionality reduction (NLDR) technique. It is a manifold learning technique which does **NOT** rely on projections, unlike PCA and random projection. In a nutshell, LLE works by first measuring how each training instance linearly relates to its nearest neighbors, and then looking for a low-dimensional representation of the training set where these local relationships are best preserved.

This approach makes it good at unrolling twisted manifolds, especially when there is not too much noise. The following code makes a Swiss roll, then uses `sklearn`'s `LocallyLinearEmbedding` class to unroll it:

The variable `t` is a 1D NumPy array containing the position of each instance along the rolled axis of the Swiss roll. We don't use it in this example, but it can be used as a target for a nonlinear regression task. The resulting 2D dataset is shown in Figure 8-10. As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the unrolled Swiss roll should be a rectangle, not this kind of stretched and twisted band. Nevertheless, LLE did a pretty good job of modeling the manifold.

Operation Principle

For each training instance $\boldsymbol{x}_{(i)}$, the algorithm identifies its k-nearest neighbours, then tries to reconstruct $\boldsymbol{x}_{(i)}$ as a linear function of these neighbors. More specifically, it tries to find the weights $w_{i,j}$ such that the squared distance between $\boldsymbol{x}_{(i)}$ and:

$$\sum_{j=1}^{m} w_{i,j} \boldsymbol{x}^{(1)}$$

is as **small as possible**, assuming $w_{i,j} = 0$ if $\boldsymbol{x}^{(i)}$ is not one of the k-nearest neighbors of $\boldsymbol{x}_{(i)}$. Thus the first step of LLE is the constrained optimization problem:

$$\hat{W} =$$

, where W is the weight matrix containing all the weights wi,j. The second constraint simply normalizes the weights for each training instance x(i).

After this step, the weight matrix W(containing the weights w i,j) encodes the local linear relationships between the training instances. The second step is to map the training instances into a d-dimensional space (where d < n) while preserving these local relationships as much as possible. If z(i) is the image of x(i) in this d-dimensional space, then we want the squared distance between z(i) and j=1

m w i,j z (j) to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 8-5. It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that Z is the matrix containing all z(i).

`sklearn`'s LLE implementation has the following computational complexity: O(m log(m)n log(k)) for finding the k-nearest neighbors, O(mnk3) for optimizing the weights, and O(dm2) for constructing the low-dimensional representations. Unfortunately, the m2 in the last term makes this algorithm scale poorly to very large datasets. As you can see, LLE is quite different from the projection techniques, and it's significantly more complex, but it can also construct much better low- dimensional representations, especially if the data is nonlinear.

Chapter 5

Unsupervised Learning

Table of Contents

5.1 Introduction

While most ML application nowadays are based on supervised learning[1], the vast majority of the available data is unlabelled:

[1] This is where companies spend most of their moneys on

> This means we have the input features (X), but do not have the labels (y). For example we can have the login information of users to a website but have no idea of their name, sex, occupation, etc.

There is a good quote by computer scientist *Yann LeCun* (Former Facebook AI Chief) given in NIPS[2] 2016 which gives a good idea on the types of learning in ML [20]:

[2] Neural Information Processing Systems

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

In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

To get a better picture, let's image a scenario. Let's say we want to create a system which will take a few pictures of each item on a manufacturing production line and detect which items are

defective. We can easily create a system which will take pictures automatically, and this might give you thousands of pictures every day. We can then build a reasonably large dataset in just a few weeks.

However we will hit a road block as there are **no labels**.

To train a regular binary classifier to predict whether an item is defective or not, will need to label every single picture either as defective or normal. This will generally require human experts to sit down and manually go through all the pictures. As we can imagine, this is rather a long, costly, and tedious task, so it will usually only be done on a small subset of the available pictures. This in turn will make the labelled dataset quite small, and the classifier's performance will be less than optimal.

> In addition, every time the company makes any change to its products, the whole process will need to be started over from scratch.

These restrictions can make ML either a tedious task at best or a massive time sink at worst. Wouldn't it be great if the algorithm could just exploit the unlabelled data without needing humans to label every picture?

> This is where unsupervised learning shows its performance.

In the previous chapter, we looked at the most common unsupervised learning task, dimensionality reduction and in this chapter we will look at a few more unsupervised tasks:

Clustering
: The goal is to group similar instances together into clusters. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and much more.

Anomaly Detection
: The goal is to learn what "normal" data looks like, and then use that to detect abnormal instances. These instances are called anomalies, or outliers, while the normal instances are called inliers. Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing, identifying new trends in time series, or removing outliers from a dataset before training another model, which can significantly improve the performance of the resulting model.[3]

[3] Anomaly detection is also known as outlier detection.

Density Estimation
: This is the task of estimating the PDF of the random process that generated the dataset. Density estimation is commonly used for anomaly detection whereby instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualisation.

5.2 Clustering Algorithms

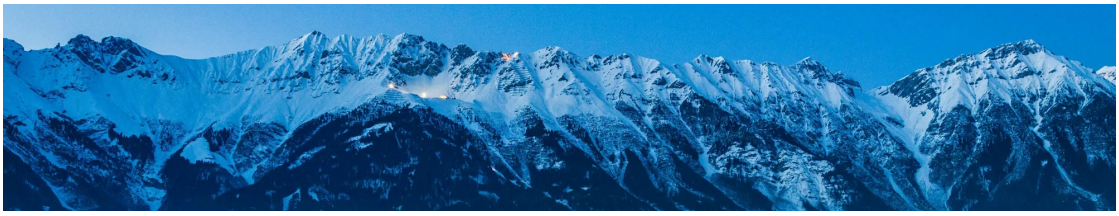


Figure 5.1: Of course, a mountain-range like this one needs no introduction in Tirol. However, for looking at flowers perhaps a snowy day is not the best.

As with all previous examples, let's use our imagination and assume we are enjoying our hike through the mountains of Tirol, perhaps somewhere in Nordkette (**Fig.** 5.1), and we stumble upon a plant we have never seen before. It could be Alpen-Edelweiß[4] or maybe something else.

[4] An illustration of Edelweiß.

We can't tell.

We look around and we notice a few more. They are not identical, however they are **sufficiently similar** for we to know that they most likely belong to the same species. We may need a botanist, or a local, to tell you what species that is, but we certainly don't need an expert to identify groups of similar-looking objects.

This is called clustering:

> It is the task of identifying similar instances and assigning them to clusters, or groups of similar instances without knowing what the instance really is.

Similar to classification, each instance gets assigned to a **group**. However, unlike classification, clustering is an unsupervised task. Consider **Fig.** 5.2: on the left is the *iris dataset*, where each instance's species[5] is represented with a different marker. It is a labelled dataset, for which classification algorithms such as logistic regression, SVMs, or random forest classifiers are well suited.

[5] i.e., its class

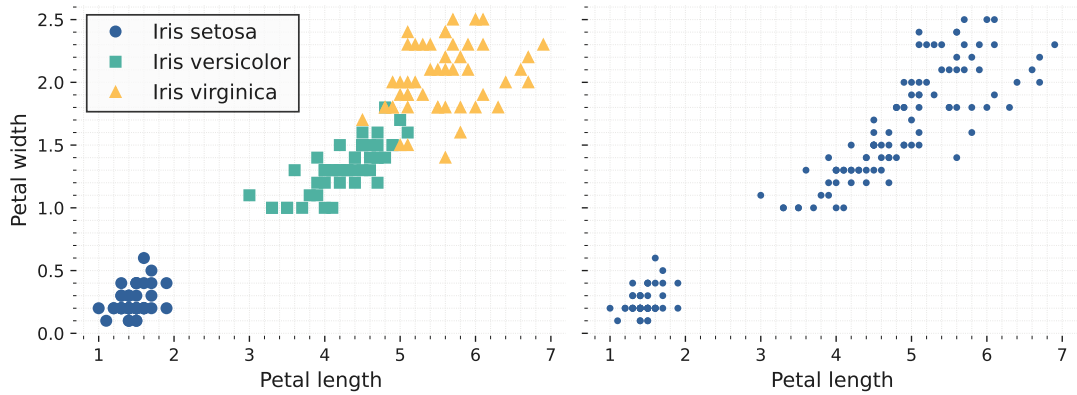


Figure 5.2: lassification (left) versus clustering (right).

On the right is the same dataset, but without the labels, so we cannot use a classification algorithm anymore. This is where clustering algorithms step in as many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two (2) distinct subclusters. That said, the dataset has two (2) additional features:

- sepal length, and
- sepal width,

which are not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well.[6]

[6] e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster.

Applications Clustering is used in a wide variety of applications, including:

Customer Segmentation

We can cluster our customers based on their purchases and their activity on our website. This is useful to understand who our customers are and what they need, so we can adapt our products and marketing campaigns to each segment [21].

> Customer segmentation can be useful in recommender systems to suggest content that other users in the same cluster enjoyed.

Data analysis

When we analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

Dimensionality reduction

Once a dataset has been clustered, it is usually possible to measure each instance's affinity with each cluster. Here, affinity is any measure of how well an instance fits into a cluster. Each instance's feature vector $\boldsymbol{x}$ can then be replaced with the vector of its cluster affinities. If there are k clusters, then this vector is k-dimensional.

> The new vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

Feature engineering

The cluster affinities can often be useful as extra features. For example, we used k-means before to add geographic cluster affinity features to the California housing dataset, and they helped us get better performance.

Anomaly detection

Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if we have clustered the users of our website based on their behavior, we can detect users with

unusual behavior, such as an unusual number of requests per second [22].

Semi-supervised learning
If we only have a few labels, we could perform clustering and propagate the labels to all the instances in the same cluster. This technique can greatly increase the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance [23].

Search engines
Some search engines let you search for images that are similar to a reference image. To build such a system, we first apply a clustering algorithm to all the images in our database. This allows similar images to end up in the same cluster. Then when a user provides a reference image, all we'd need to do is use the trained clustering model to find this image's cluster, and we could then simply return all the images from this cluster [24].

Image segmentation
By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the number of different colors in an image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object [25].

There is **no universal definition of what a cluster is** as it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms, for example, look for instances centered around a particular point, called a **centroid**. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section of our chapter, we will look at two (2) popular clustering algorithms:

- k-means, and
- DBSCAN,

and explore some of their applications, such as non-linear dimensionality reduction, semi-supervised learning, and anomaly detection.

5.2.1 k-means

Consider the unlabelled dataset represented in **Fig.** 5.3. It is clear to us to say we can clearly see five (5) blobs of instances. The *k*-means algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations. It was proposed by *Stuart Lloyd* at Bell Labs in 1957 as a technique for Pulse Code Modulation (PCM), but it was only published outside of the company in 1982 [26]. In 1965, *Edward W. Forgy* had published virtually the same algorithm [27], so *k*-means is sometimes referred to as the Lloyd-Forgy algorithm.

Let's train a *k*-means clusterer on this dataset. It will try to find each blob's center and assign each

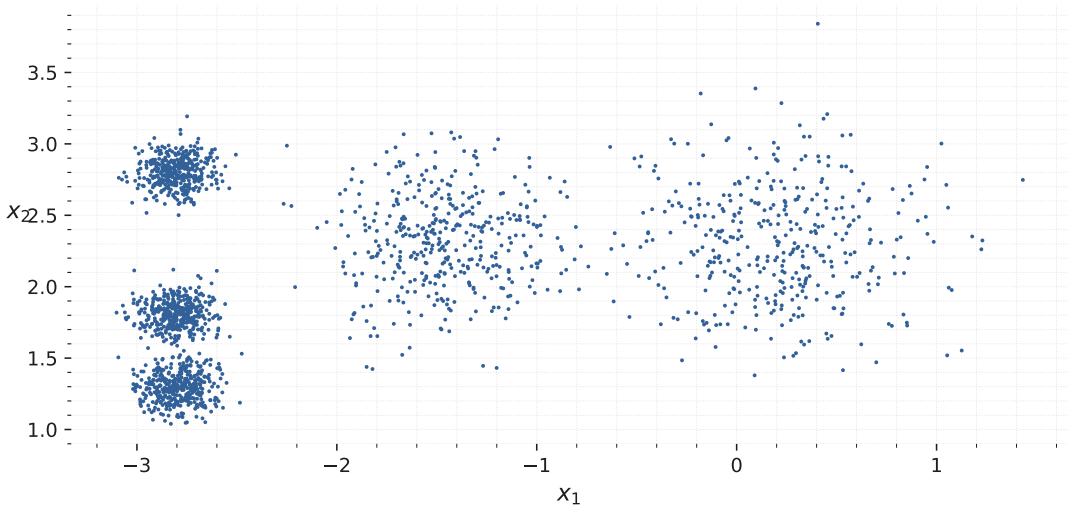


Figure 5.3: An unlabelled dataset composed of five blobs of instances.

instance to the closest blob:

C.R. 1
python
```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

blob_centers = np.array([[ 0.2,  2.3], [-1.5 ,  2.3], [-2.8,  1.8],
                         [-2.8,  2.8], [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
X, y = make_blobs(n_samples=2000, centers=blob_centers, cluster_std=blob_std,
                  random_state=42)

k = 5
kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
y_pred = kmeans.fit_predict(X)
```

> For the method to work, we have to specify the number of clusters (k) which the algorithm must find.

In this example, as said previously, it is obvious k should be set to five (5), but in general it is not that easy. Each instance will be assigned to one of the five (5) clusters. In the context of clustering, an instance's label is the index of the cluster to which the algorithm assigns this instance.

This should not to be confused with the class labels in classification, which are used as targets.[7]

[7] remember, clustering is an unsupervised learning task

The `KMeans` instance preserves the predicted labels of the instances it was trained on, available via the `labels_` instance variable:

C.R. 2
python
```
print(y_pred)
```

C.R. 3
```
[2 1 3 ... 1 2 0]
```
text

C.R. 4
```
y_pred is kmeans.labels_
```
python

C.R. 5
```
True
```
text

We can also take a look at the five (5) centroids that the algorithm found:

C.R. 6
```
print(kmeans.cluster_centers_)
```
python

C.R. 7
```
[[-2.80372723  1.80873739]
 [ 0.20925539  2.30351618]
 [-2.79846237  2.80004584]
 [-1.4453407   2.32051326]
 [-2.79244799  1.2973862 ]]
```
text

We can easily assign new instances to the cluster whose centroid is closest:

C.R. 8
```
import numpy as np

X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
print(kmeans.predict(X_new))
```
python

C.R. 9
```
[1 1 2 2]
```
text

If we plot the cluster's decision boundaries, we get a Voronoi tessellation[8] which can be seen in **Fig.** 5.4, where each centroid is represented with an X.

[8] a type of tessellation pattern in which a number of points scattered on a plane subdivides in exactly n cells enclosing a portion of the plane that is closest to each point.

The vast majority of the instances were clearly assigned to the appropriate cluster, but a good part of the instances were mislabelled, Such as parts in the lower left where there is obviously two (2) centre points, but the algorithm decided there is only one. Indeed, the k-means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the **centroid**.

Instead of assigning each instance to a single cluster, which is called **hard clustering**, it can be useful to give each instance a score per cluster, which is called soft clustering.[9] The score can be the distance between the instance and the centroid or a similarity score, such as the Gaussian RBF. In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

[9] This is a similar behaviour to hard v. soft voting we encountered in Random Forest.

C.R. 10
```
print(kmeans.transform(X_new).round(2))
```
python

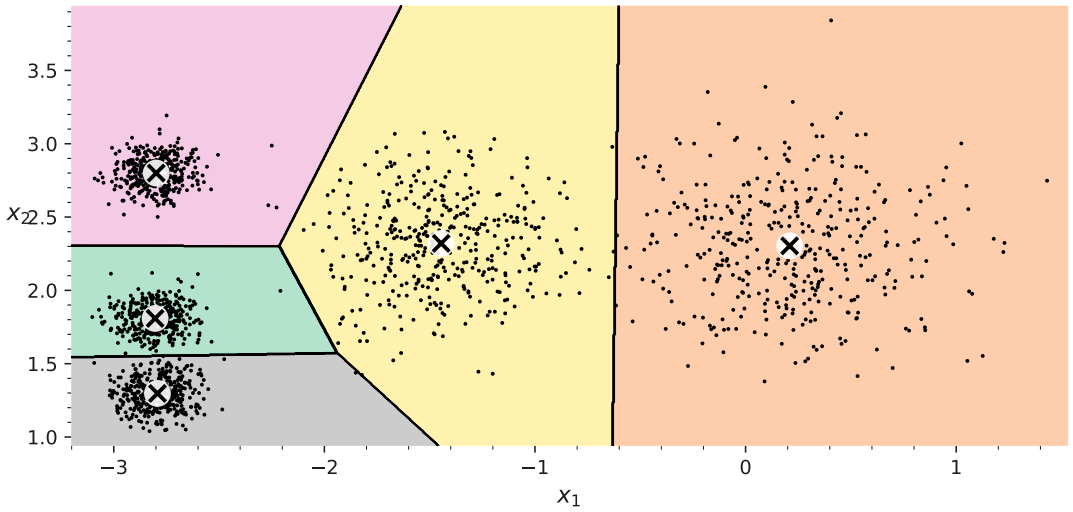


Figure 5.4: *k*-means decision boundaries (Voronoi tessellation)

```
[[2.81 0.37 2.91 1.48 2.88]
 [5.81 2.81 5.85 4.46 5.83]
 [1.21 3.28 0.28 1.7  1.72]
 [0.72 3.22 0.36 1.56 1.22]]
```

text

In the aforementioned example, the first instance in `X_new` is located at a distance of about 2.84 from the 1st centroid, 0.59 from the 2nd centroid, 1.5 from the 3rd centroid, 2.9 from the 4th centroid, and 0.31 from the 5th centroid.

If we have a high-dimensional dataset and we transform it this way, we end up with a k-dimensional dataset. This transformation can be a very efficient non-linear dimensionality reduction technique. Alternatively, we can use these distances as extra features to train another model.

The Operation Principle

Let's try to understand *k*-means via an example. Suppose we were given the centroids. We could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Or, if we were given all the instance labels, we could easily locate each cluster's centroid by computing the mean of the instances in that cluster.

But here are given neither the labels nor the centroids, so how can we proceed?

We start by placing the centroids randomly.[10] Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving.

[10] e.g., by picking *k* instances at random from the dataset and using their locations as centroids.

The algorithm is **guaranteed** to converge in a finite number of steps.

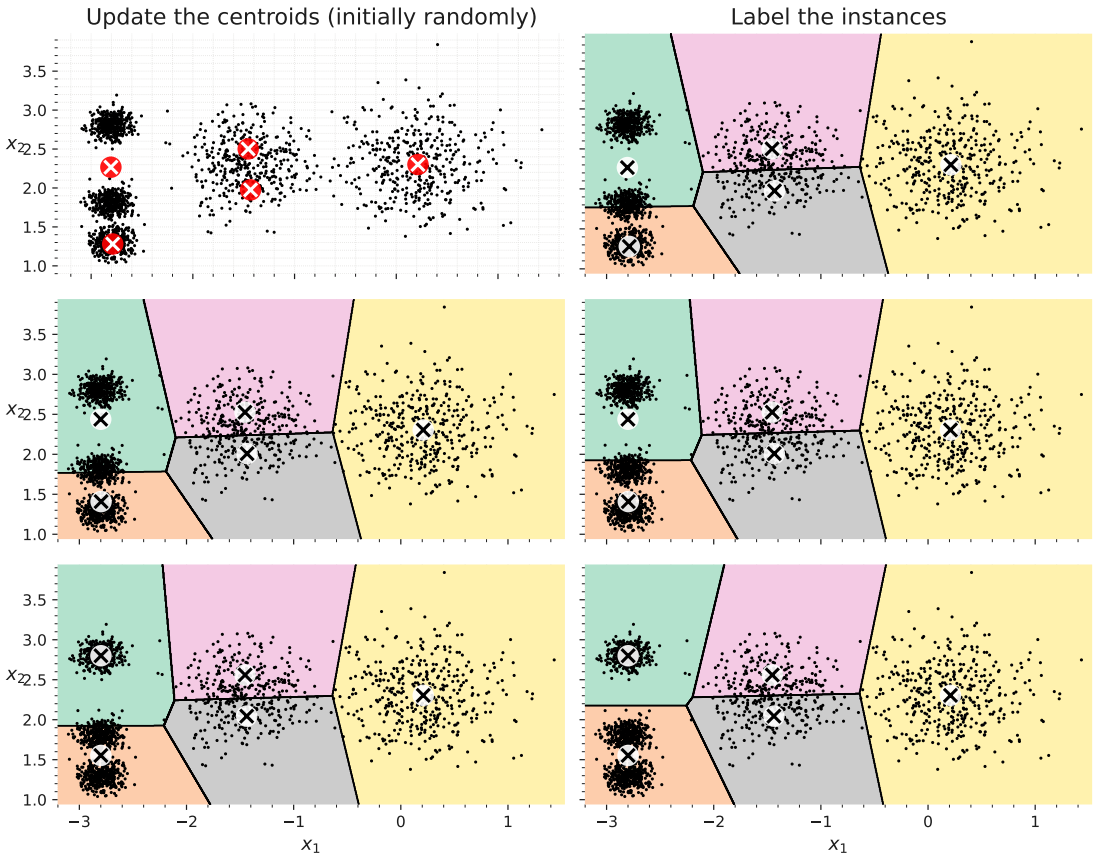


Figure 5.5: The k-means algorithm.

That's because the mean squared distance between the instances and their closest centroids can only go down at each step, and since it cannot be negative, it's guaranteed to converge. We can see the algorithm in action in **Fig.** 5.5:

Let's try to explain the behaviour of the figure.

1. the centroids are initialised randomly (top left)
2. then the instances are labelled (top right),
3. the centroids are updated (center left),
4. the instances are relabelled (center right),

and so on. As we can see, in just three (3) iterations the algorithm has reached a clustering that seems close to optimal.

Information: Computational Complexity

The computational complexity of the algorithm is **generally linear** with regards to the number of instances (m), the number of clusters (k), and the number of dimensions (n). However, this is only true when the data has a clustering structure. If it does not, then in the worst-case scenario the complexity can increase **exponentially** with the number of instances.

In practice, this rarely happens, and k-means is generally one of the fastest clustering algorithms.

Although the algorithm is guaranteed to converge, it may not converge to the right solution:[11] whether it does or not depends on the centroid initialisation. **Fig.** 5.6 shows two (2) sub-optimal solutions that the algorithm can converge to if we are not lucky with the random initialisation step.

[11] i.e., it may converge to a local optimum, instead of the global.

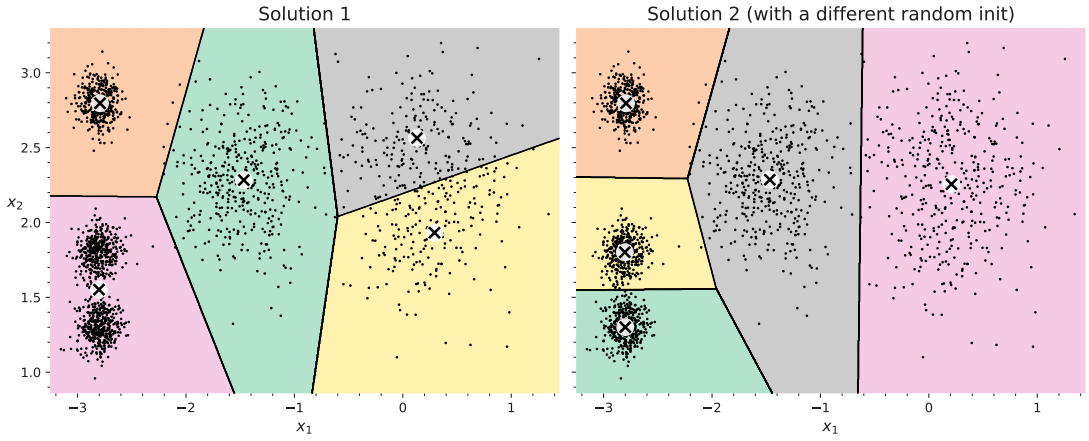


Figure 5.6: Suboptimal solutions due to unlucky centroid initialisation.

Let's take a look at a few ways we can mitigate this risk by improving the centroid initialisation.

Centroid initialisation methods

If we happen to know approximately where the centroids should be,[12] then we can set the `init` hyperparameter to a **numpy** array containing the list of centroids, and set `n_init` to `1`:

[12] i.e., if we ran another clustering algorithm earlier.

C.R. 12

```python
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5,
                init=good_init,
                n_init=1,
                random_state=42)
kmeans.fit(X)
```

Another solution is to run the algorithm multiple times with different random initialisations and keep the best solution. The number of random initialisation is controlled by the `n_init` hyperparameter:

by default it is equal to 10, which means that the whole algorithm described earlier runs

10 times when we call `fit()`, and `sklearn` keeps the best solution.

But how exactly does it know which solution is the best? Well, it uses a performance metric. That metric is called the **model's inertia**, which is the sum of the squared distances between the instances and their closest centroids.

$$\sum_{i=1}^{N} \left(x_i - C_k\right)^2$$

where N is the number of samples, x is the value of a sample, C is the centre of the cluster centroid. For our example, this value is roughly equal to:

- 219.4 for the model on the left in **Fig.** 5.12,
- 258.6 for the model on the right in **Fig.** 5.12,
- 211.6 for the model in **Fig.** 5.4.

The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia.

In this example, the model in **Fig.** 5.4 will be selected.[13] For the curious, a model's inertia is accessible via the `inertia_` instance variable:

[13] unless we are very unlucky with `n_init` consecutive random initialisation

C.R. 13
```python
kmeans.inertia_
```

C.R. 14
```text
211.59853725816836
```

The `score()` method returns the negative inertia:[14]

[14] it's negative because a predictor's score() method must always respect `sklearn`'s "greater is better" rule: if a predictor is better than another, its `score()` method should return a greater score

C.R. 15
```python
kmeans.score(X)
```

C.R. 16
```text
-211.5985372581684
```

An important improvement to the k-means algorithm, k-means++, was proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii [28]. They introduced a smarter initialisation step that tends to select centroids that are distant from one another, and this improvement makes the k-means algorithm much less likely to converge to a sub-optimal solution.

The paper showed, the additional computation required for the smarter initialisation step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution.

The *k*-means++ initialisation works as follows:

1. Take one centroid $\boldsymbol{c}^{(1)}$, chosen uniformly at random from the dataset,
2. Take the new centroid $\boldsymbol{c}^{(i)}$, choosing an instance $\boldsymbol{x}^{(i)}$ with probability:

$$p\left(\boldsymbol{x}^{(i)}\right) = \frac{D\left(\boldsymbol{x}^{(i)}\right)^2}{\sum_{j=1}^{m} D\left(\boldsymbol{x}^{(i)}\right)^2}$$

   where $D\left(\boldsymbol{x}^{(i)}\right)^2$ is the distance between the instance $\boldsymbol{x}^{(i)}$ and the closest centroid that was already chosen.

   > This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.

3. Repeat the previous step until all *k* centroids have been chosen.

The `KMeans` class uses this initialisation method by default. To force it to use the original method (i.e., picking *k* instances randomly to define the initial centroids), then we can set the init hyperparameter to `"random"`.

> We will rarely need to do this.

Accelerated and mini-batch

Another improvement to the *k*-means algorithm was proposed in a 2003 paper by *Charles Elkan* [29]. On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations. Elkan achieved this by exploiting the triangle inequality[15] and by keeping track of lower and upper bounds for distances between instances and centroids. However, Elkan's algorithm does not always accelerate training, and sometimes it can even slow down training significantly as it depends on the dataset.

[15] The triangle inequality is AC ≤ AB + BC, where A, B and C are three points and AB, AC, and BC are the distances between these points.

> To give it a try, set `algorithm="elkan"`.

Yet another important variant of the *k*-means algorithm was proposed in a 2010 paper by David Sculley [30]. Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm[16] and makes it possible to cluster huge datasets that do not fit in memory. `sklearn` implements this algorithm in the `MiniBatchKMeans` class, which we can use just like the `KMeans` class:

[16] typically by a factor of three to four

C.R. 17

```python
from sklearn.cluster import MiniBatchKMeans

```

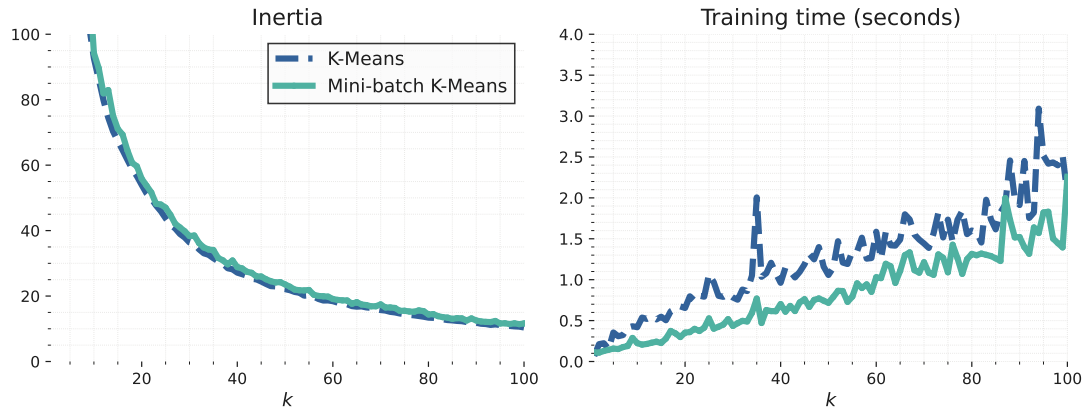


Figure 5.7: Mini-batch k-means has a higher inertia than k-means (left) but it is much faster (right), especially as k increases.

C.R. 18

```python
minibatch_kmeans = MiniBatchKMeans(n_clusters=10, batch_size=10,
↪  n_init=3, random_state=42)
minibatch_kmeans.fit(X_memmap)
```

If the dataset does not fit in memory, the simplest option is to use the `memmap` class. Alternatively, we can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, as we will need to perform multiple initialisations and select the best one ourselves.

Although the mini-batch k-means algorithm is much faster than the regular k-means algorithm, its inertia is generally slightly worse. We can see this in **Fig.** 5.7. The plot on the left compares the inertiae of mini-batch k-means and regular k-means models trained on the previous five-blobs dataset using various numbers of clusters k. The difference between the two curves is small, but visible. In the plot on the right, we can see that mini-batch k-means is roughly 1.5 - 2 times faster than regular k-means on this dataset.

Finding the optimal number of clusters

So far, we've set the number of clusters k to five (5) as it was obvious by looking at the data that this was the correct number of clusters. But in general, it won't be so easy to know how to set k, and the result might be quite bad if we set it to the wrong value. As we can see in **Fig.** 5.8, for this dataset setting k to 3 or 8 results in fairly bad models.

We might be thinking that we could just pick the model with the lowest inertia. Unfortunately, it is not that simple. The inertia for $k = 3$ is about 653.2, which is much higher than for $k = 5$ with a value of 211.6. But with $k = 8$, the inertia is just 119.1.

> The inertia is not a good performance metric when trying to choose k as it keeps getting lower as we increase k.

Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. To see this visually Let's plot the inertia as a function of k. When we do this, the curve often contains an inflexion point called the elbow (see **Fig.** 5.9).

As we can see, the inertia drops very quickly as we increase k up to 4, but then it decreases much more slowly as we keep increasing k. This curve has roughly the shape of an arm, and there is an elbow at $k = 4$. So, if we did not know better, we might think 4 was a good choice as any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.

This technique for choosing the best value for the number of clusters is rather coarse. A more precise[17] approach is to use the silhouette score, which is the mean silhouette coefficient over all the instances. An instance's silhouette coefficient is equal to:

[17] but also more computationally expensive

$$\frac{b - a}{\max(a, b)}$$

where a is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and b is the mean nearest-cluster distance[18]. The silhouette coefficient can vary between -1 and +1. A coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary and finally, a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

[18] the mean distance to the instances of the next closest cluster, defined as the one that minimizes b, excluding the instance's own cluster

To compute the silhouette score, we can use `sklearn`'s `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

C.R. 19

```python
from sklearn.metrics import silhouette_score

silhouette_score(X, kmeans.labels_)
```

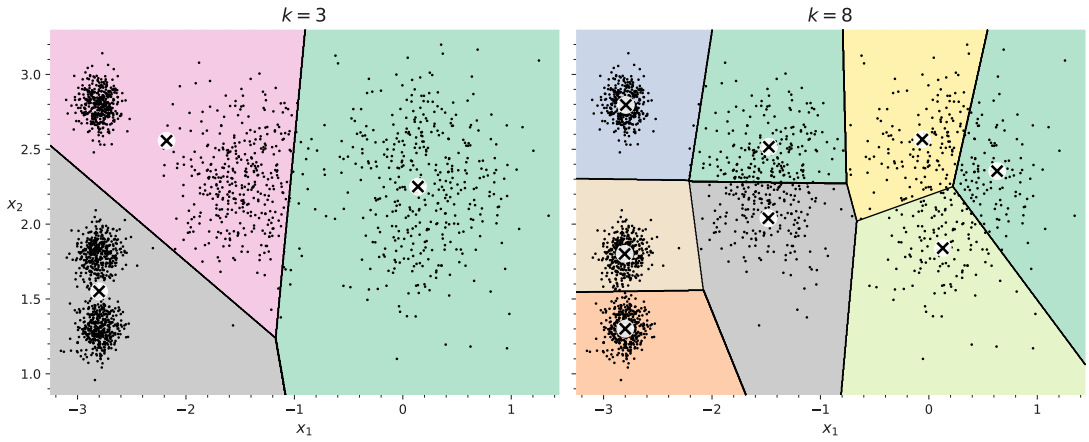


Figure 5.8: Bad choices for the number of clusters: when k is too small, separate clusters get merged (left), and when k is too large, some clusters get chopped into multiple pieces (right)

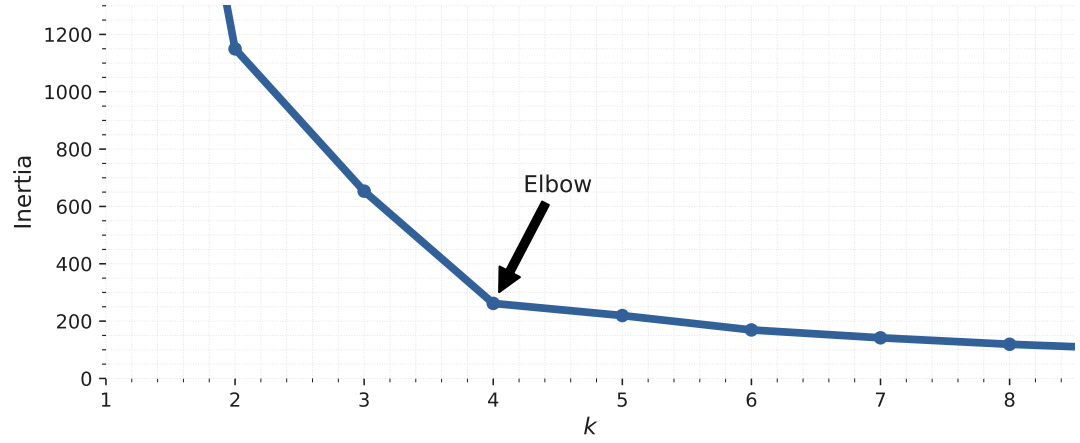


Figure 5.9: Plotting the inertia as a function of the number of clusters k

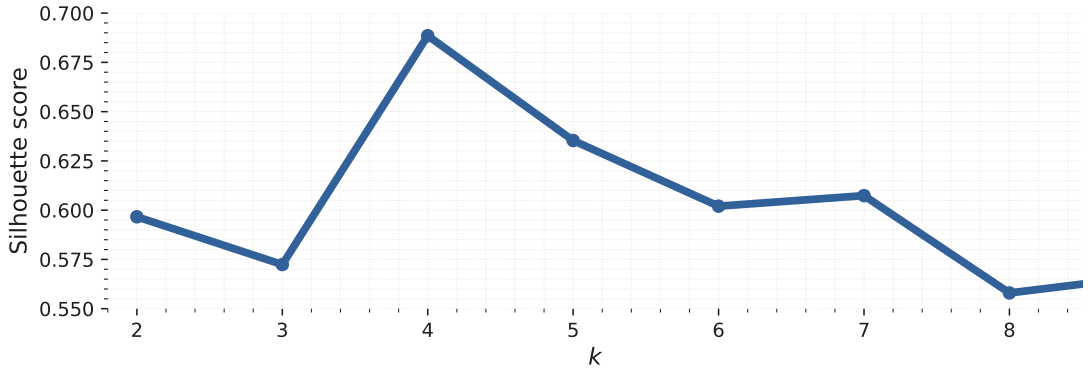


Figure 5.10: Selecting the number of clusters k using the silhouette score.

```
0.655517642572828
```
text

Let's compare the silhouette scores for different numbers of clusters (see **Fig.** 5.10).

As we can see, this visualisation gives more information compared to the previous one:

> although it confirms that $k = 4$ is a very good choice, it also highlights the fact that $k = 5$ is quite good as well, at least much better than $k = 6$ or 7.

> This was not visible when comparing inertiae.

An even more informative visualisation is obtained when we plot every instance's silhouette coefficient, sorted by the clusters they are assigned to and by the value of the coefficient. This is called a silhouette diagram (see **Fig.** 5.11). Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances in the cluster, and its width represents the sorted silhouette coefficients of the instances in the cluster[19].

[19] wider is better.

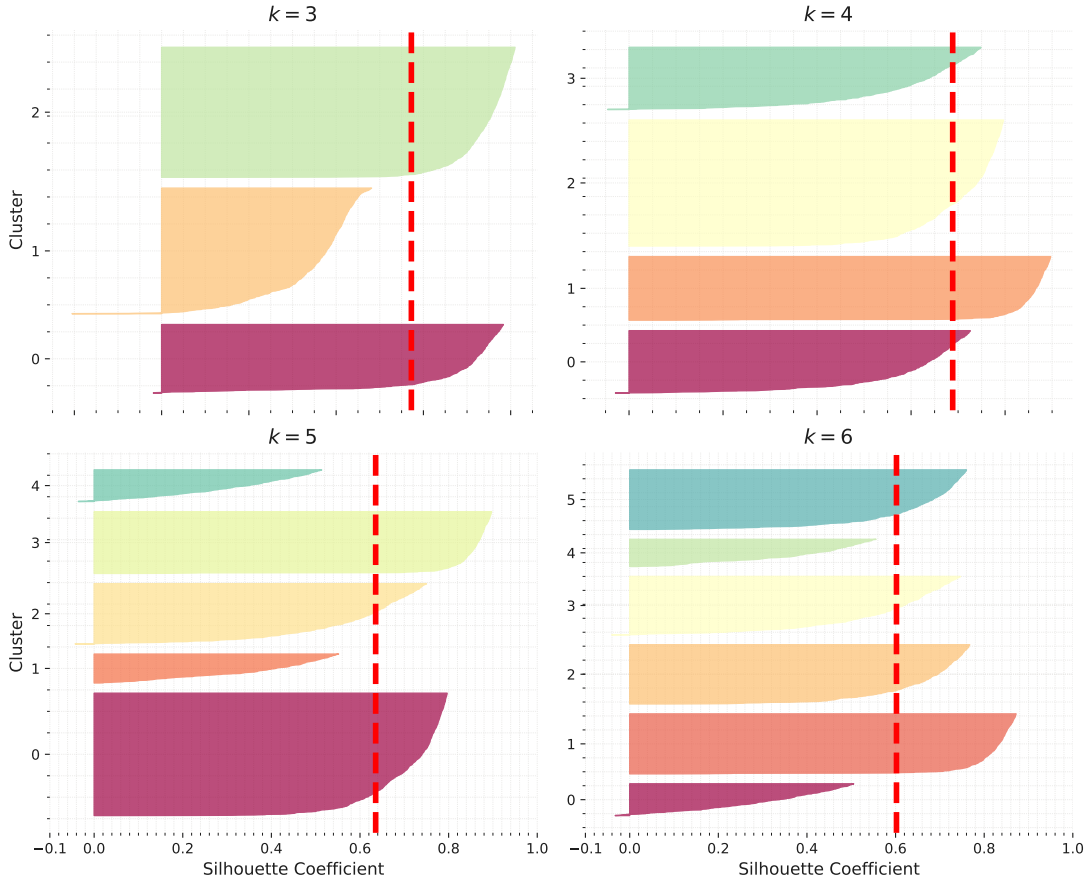


Figure 5.11: Analyzing the silhouette diagrams for various values of k.

The vertical dashed lines represent the mean silhouette score for each number of clusters. When most instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. Here we can see that when k = 3 or 6, we get bad clusters. But when k = 4 or 5, the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0.

When k = 4, the cluster at index 2 (the second from the top) is rather big. When k = 5, all clusters have similar sizes. So, even though the overall silhouette score from k = 4 is slightly greater than for k = 5, it seems like a good idea to use k = 5 to get clusters of similar sizes.

5.2.2 Limits of K-Means

Despite its many merits, most notably being fast and scalable, *k*-means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid sub-optimal solutions, plus we need to specify the number of clusters, which can be quite a hassle. Moreover, *k*-means does not behave very well when the clusters have varying sizes, different densities, or non-spherical shapes. For example,

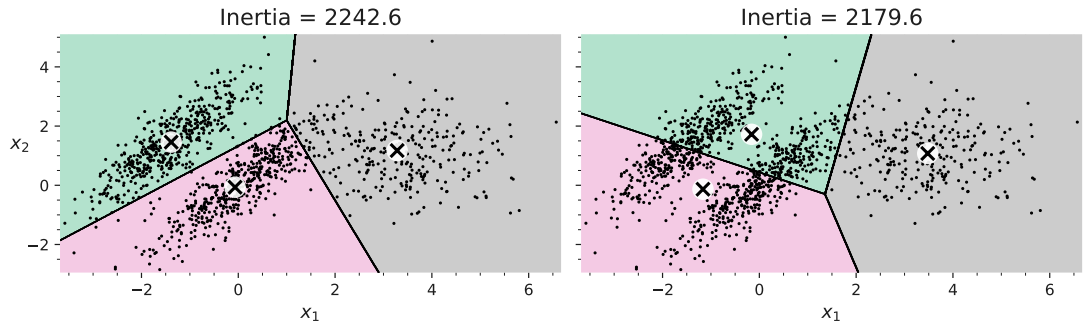


Figure 5.12: *k*-means fails to cluster these ellipsoidal blobs properly.

Fig. 5.12 shows how *k*-means clusters a dataset containing three (3) ellipsoidal clusters of different dimensions, densities, and orientations.

As can be seen, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better. On these types of elliptical clusters, **Gaussian mixture models** work great.

> It is important to scale the input features before we run *k*-means, or the clusters may be very stretched and *k*-means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally helps *k*-means.

Now let's look at a few ways we can benefit from clustering and for these will use *k*-means

5.2.3 Using Clustering for Image Segmentation

Image segmentation is the task of partitioning an image into multiple segments. There are several variants: In color segmentation,

Colour Segmentation pixels with a similar color get assigned to the same segment. This is sufficient in many applications.

> For example, if we want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.

Semantic Segmentation all pixels that are part of the same object type get assigned to the same segment.

> For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the pedestrian segment.

Instance Segmentation all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian.

The state of the art in semantic or instance segmentation today is achieved using complex architectures based on CNN[20]. Here we are going to focus on the colour segmentation task, using *k*-means. We'll start by importing the Pillow package, which we'll then use to load the `Fruit.png` image (see the upper-left image in **Fig.** 5.13), assuming it's located at filepath:

[20] Which is taught in **B.Sc. Image processing**.

C.R. 21
```python
import PIL

image = np.asarray(PIL.Image.open(filepath))
print(image.shape)
```

C.R. 22
```text
(680, 680, 3)
```

The image is represented as a 3D array. The first dimension's size is the height, the second is the width, and the third is the number of color channels, in this case red, green, and blue (RGB). In other words, for each pixel there is a 3D vector containing the intensities of red, green, and blue as unsigned 8- bit integers between 0 and 255. Some images may have fewer channels[21], and some images may have more channels (such as images with an additional alpha channel for transparency, or satellite images, which often contain channels for additional light frequencies (like infrared). The following code reshapes the array to get a long list of RGB colors, then it clusters these colours using *k*-means with eight clusters. It creates a `segmented_img` array containing the nearest cluster centre for each pixel (i.e., the mean colour of each pixel's cluster), and lastly it reshapes this array to the original image shape. The third line uses advanced NumPy indexing; for example, if the first 10 labels in `kmeans_.labels_` are equal to 1, then the first 10 colors in `segmented_img` are equal to `kmeans.cluster_centers_[1]`:

[21] such as grayscale images, which only have one.

C.R. 23
```python
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8, n_init=10, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)

segmented_imgs = []
n_colors = (10, 8, 6, 4, 2)

for n_clusters in n_colors:
    kmeans = KMeans(n_clusters=n_clusters, n_init=10, random_state=42).fit(X)
    segmented_img = kmeans.cluster_centers_[kmeans.labels_]
    segmented_imgs.append(segmented_img.reshape(image.shape))

```

This outputs the image shown in the upper right of **Fig.** 5.13. We can experiment with various numbers of clusters, as shown in the figure. When we use fewer than eight clusters, notice that the ladybug's flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is because *k*-means prefers clusters of similar sizes. The ladybug is small-much

smaller than the rest of the image-so even though its colour is flashy, *k*-means fails to dedicate a cluster to it.

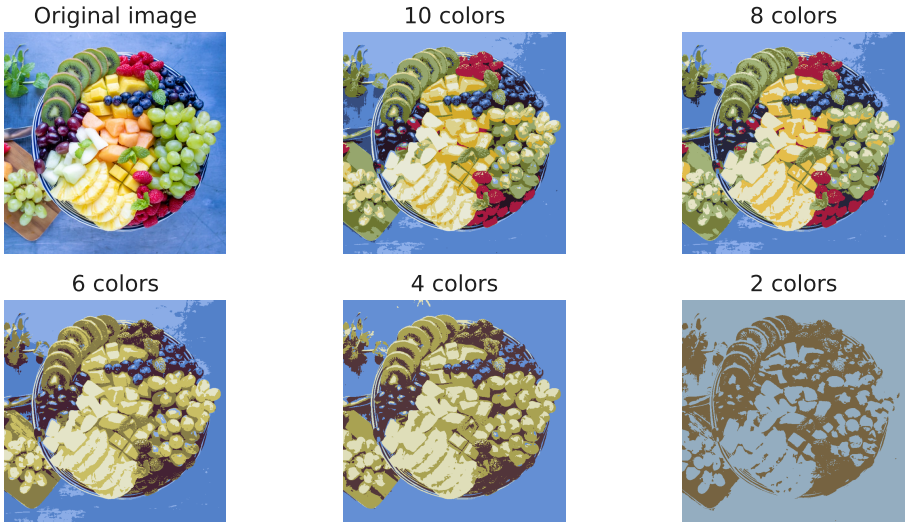


Figure 5.13: Image segmentation using *k*-means with various numbers of color clusters

Now this looks very pretty. Now it is time to look at another application of clustering.

5.2.4 Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabelled instances and very few labelled instances. In this section, we'll use the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale 8-by-8 images representing the digits 0 to 9. First, let's load and split the dataset (it's already shuffled):

C.R. 24
```python
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]
```

We will pretend we only have labels for 50 instances. To get a baseline performance, let's train a logistic regression model on these 50 labelled instances:

C.R. 25
```python
from sklearn.linear_model import LogisticRegression

n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

We can then measure the accuracy of this model on the test set:

Figure 5.14: Fifty representative digit images (one per cluster).

> The test set must be labelled:

C.R. 26
```python
log_reg.score(X_test, y_test)
```

C.R. 27
```text
0.7581863979848866
```

The model's accuracy is just 75.8%. That's not great: indeed, if we try training the model on the full training set, we will find that it will reach about 90.9% accuracy. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then, for each cluster, we'll find the image closest to the centroid. We'll call these images the representative images where we can see 50 of them in **Fig.** 5.14

C.R. 28
```python
k = 50
kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = X_digits_dist.argmin(axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Let's look at each image and manually label them:

C.R. 29
```python
y_representative_digits = np.array([
    8, 4, 9, 6, 7, 5, 3, 0, 1, 2,
    3, 3, 4, 7, 2, 1, 5, 1, 6, 4,
    5, 6, 5, 7, 3, 1, 0, 8, 4, 7,
    1, 1, 8, 2, 9, 9, 5, 9, 7, 4,
    4, 9, 7, 8, 2, 6, 6, 3, 2, 8
])
```

Now we have a dataset with just 50 labelled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

C.R. 30
```python
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_representative_digits, y_representative_digits)
log_reg.score(X_test, y_test)
```

C.R. 31
```text
0.8387909319899244
```

Wow! We jumped from 75.8% accuracy to 83.8%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances. But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called label propagation:

C.R. 32
```python
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]
```

Now let's train the model again and look at its performance:

C.R. 33
```python
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train, y_train_propagated)
```

C.R. 34
```python
log_reg.score(X_test, y_test)
```

C.R. 35
```text
0.8589420654911839
```

We got another significant accuracy boost.

Active Learning

Information: Active Learning

To continue improving our model and our training set, the next step could be to do a few rounds of active learning, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them. There are many different strategies for active learning, but one of the most common ones is called **uncertainty sampling**. Here is how it works:

1. The model is trained on the labelled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
2. The instances for which the model is most uncertain(i.e.,where its estimated probability is lowest) are given to the expert for labelling.
3. We iterate this process until the performance improvement stops being worth the labeling effort.

Other active learning strategies include labeling the instances that would result in the largest model change or the largest drop in the model's validation error, or the instances that different models disagree on (e.g., an SVM and a random forest).

Before we move on to Gaussian mixture models, let's take a look at Density-Based Spatial Clustering of Applications with Noise (DBSCAN), another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

5.2.5 DBSCAN

The DBSCAN algorithm defines clusters as continuous regions of high density. Here is how it works:

1. For each instance, the algorithm counts how many instances are located within a small distance ϵ from it. This region is called the instance's ϵ-neighbourhood.

2. If an instance has at least `min_samples` instances in its ϵ-neighborhood (including itself), then it is considered a core instance. In other words, core instances are those that are located in dense regions.

3. All instances in the neighborhood of a core instance belong to the same cluster. This neighbourhood may include other core instances, therefore, a long sequence of neighboring core instances forms a single cluster.

4. Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

This algorithm works well if all the clusters are well separated by low-density regions. The DBSCAN class in `sklearn` is as simple to use as we might expect. Let's test it on the moons dataset, which is a toy dataset:

C.R. 36
python
```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

The labels of all the instances are now available in the `labels_` instance variable:

C.R. 37
python
```
print(dbscan.labels_[:10])
```

C.R. 38
text
```
[ 0  2 -1 -1  1  0  0  0  2  5]
```

Notice that some instances have a cluster index equal to -1, which means that they are considered as anomalies by the algorithm. The core instances indices are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

C.R. 39 python
```python
print(dbscan.core_sample_indices_[:10])
```

C.R. 40 text
```
[ 0  4  5  6  7  8 10 11 12 13]
```

C.R. 41 python
```python
print(dbscan.components_)
```

C.R. 42 text
```
[[-0.02137124  0.40618608]
 [-0.84192557  0.53058695]
 [ 0.58930337 -0.32137599]
 ...
 [ 1.66258462 -0.3079193 ]
 [-0.94355873  0.3278936 ]
 [ 0.79419406  0.60777171]]
```

This clustering is represented in the Left Hand Side (LHS) plot of **Fig.** 5.15. As we can see, it identified quite a lot of anomalies, plus seven different clusters. Fortunately, if we widen each instance's neighbourhood by increasing eps to 0.2, we get the clustering on the right, which looks much better. Let's continue with this model.

For example, let's train a `KNeighborsClassifier`:

C.R. 43 python
```python
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

Now, given a few new instances, we can predict which clusters they most likely belong to and even estimate a probability for each cluster:

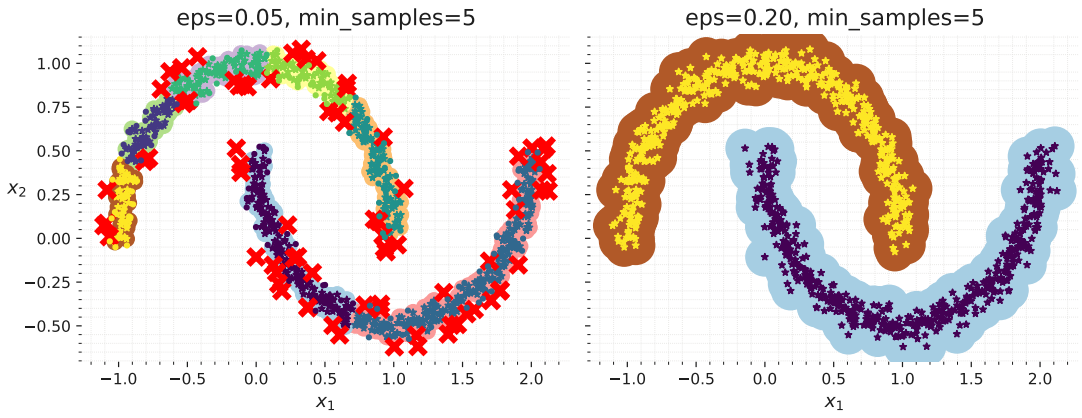


Figure 5.15: DBSCAN clustering using two different neighborhood radiuses.

C.R. 44
```python
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
print(knn.predict(X_new))
```

C.R. 45
```text
[1 0 1 0]
```

C.R. 46
```python
print(knn.predict_proba(X_new))
```

C.R. 47
```text
[[0.18 0.82]
 [1.   0.  ]
 [0.12 0.88]
 [1.   0.  ]]
```

Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies.

> This choice depends on the final task

The decision boundary is represented in **Fig.** 5.16 (the crosses represent the four instances in `X_new`). Notice that since there is no anomaly in the training set, the classifier always chooses a cluster, even

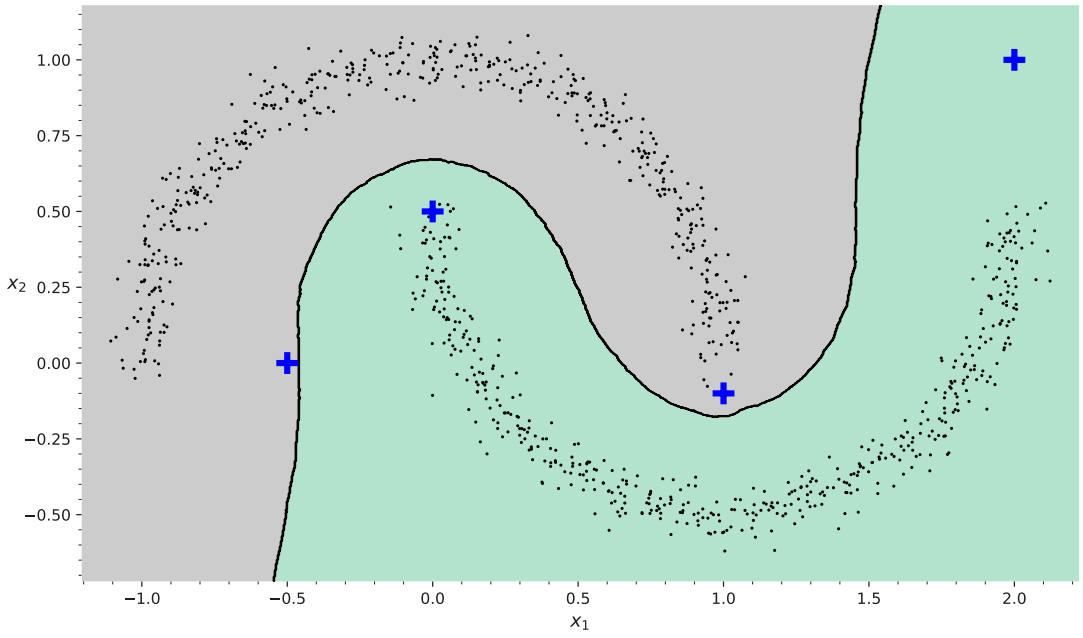


Figure 5.16: Decision boundary between two clusters

when that cluster is far away. It is fairly straightforward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it returns

the distances and the indices of the k-nearest neighbours in the training set (two matrices, each with k columns):

C.R. 48
```python
y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
y_pred[y_dist > 0.2] = -1
print(y_pred.ravel())
```

C.R. 49
```text
[-1  0  1 -1]
```

In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`). If the density varies significantly across the clusters, however, or if there's no sufficiently low- density region around some clusters, DBSCAN can struggle to capture all the clusters properly. Moreover, its computational complexity is roughly O(m2n), so it does not scale well to large datasets.

Other Clustering Algorithms

`sklearn` implements several more clustering algorithms that we should take a look at. Here is just some of them:

Agglomerative clustering A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If we drew a tree with a branch for every pair of clusters that merged, we would get a binary tree of clusters, where the leaves are the individual instances. This approach can capture clusters of various shapes; it also produces a flexible and informative cluster tree instead of forcing we to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if we provide a connectivity matrix, which is a sparse m Œ m matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

BIRCH The balanced iterative reducing and clustering using hierarchies (BIRCH) algorithm was designed specifically for very large datasets, and it can be faster than batch k-means, with similar results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory while handling huge datasets.

Mean-shift This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles

stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one-the radius of the circles, called the bandwidth), and it relies on local density estimation. But unlike DBSCAN, mean-shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is O(m2n), so it is not suited for large datasets.

Affinity Propagation In this algorithm, instances repeatedly exchange messages between one another until every instance has elected another instance (or itself) to represent it. These elected instances are called exemplars. Each exemplar and all the instances that elected it form one cluster. In real-life politics, we typically want to vote for a candidate whose opinions are similar to ours, but we also want them to win the election, so we might choose a candidate we don't fully agree with, but who is more popular. We typically evaluate popularity through polls. Affinity propagation works in a similar way, and it tends to choose exemplars located near the center of clusters, similar to *k*-means. But unlike with *k*-means, we don't have to pick a number of clusters ahead of time: it is determined during training. Moreover, affinity propagation can deal nicely with clusters of different sizes. Sadly, this algorithm has a computational complexity of O(m2), so it is not suited for large datasets.

Spectral clustering This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low- dimensional space (`sklearn`'s implementation uses *k*-means). Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Now let's dive into Gaussian mixture models, which can be used for density estimation, clustering, and anomaly detection.

5.3 Gaussian Mixtures

A Gaussian Mixture Model (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density, and orientation, just like in **Fig.** 5.8. When we observe an instance, we know it was generated from one of the Gaussian distributions, but we are not told which one, and we do not know what the parameters of these distributions are.

There are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class, we must know in advance the number k of Gaussian distributions. The dataset X is assumed to have been generated through the following probabilistic process:

- For each instance, a cluster is picked randomly from among k clusters. The probability of choosing the j_{th} cluster is the cluster's weight $\phi^{(j)}$. The index of the cluster chosen for the j_{th} instance is noted $z^{(i)}$.
- If the i^{th} instance was assigned to the j_{th} cluster (i.e., $z^{(i)} = j$), then the location $\boldsymbol{x}^{(i)}$ of this instance is sampled randomly from the Gaussian distribution with mean $\mu^{(j)}$ and covariance matrix $\Sigma^{(j)}$. This is noted as:

$$\boldsymbol{x}^{(i)} \sim \mathcal{N}\left(\mu^{(j)}, \Sigma^{(j)}\right)$$

So what can we do with such a model? Well, given the dataset $\boldsymbol{X}$, we typically want to start by estimating the weights ϕ and all the distribution parameters $\mu^{(1)}$ to $\mu^{(k)}$ and $\Sigma^{(1)}$ to $\Sigma^{(k)}$. `sklearn`'s `GaussianMixture` class makes this super easy:

C.R. 50 python
```python
from sklearn.mixture import GaussianMixture
```

C.R. 51 python
```python
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

C.R. 52 python
```python
print(gm.weights_)
```

C.R. 53 text
```
[0.40005972 0.20961444 0.39032584]
```

C.R. 54 python
```python
print(gm.means_)
```

C.R. 55
```
[[-1.40764129  1.42712848]
 [ 3.39947665  1.05931088]
 [ 0.05145113  0.07534576]]
```
text

Great, it worked fine! Indeed, two of the three clusters were generated with 500 instances each, while the third cluster only contains 250 instances. So the true cluster weights are 0.4, 0.2, and 0.4, respectively, and that's roughly what the algorithm found. Similarly, the true means and covariance matrices are quite close to those found by the algorithm. But how? This class relies on the Expectation Maximisation (EM) algorithm, which has many similarities with the *k*-means algorithm where it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the expectation step) and then updating the clusters (this is called the maximisation step). In the context of clustering, we can think of EM as a generalisation of *k*-means that not only finds the cluster centres ($\mu^{(1)}$ to $\mu^{(k)}$), but also their size, shape, and orientation ($\Sigma^{(1)}$ to $\Sigma^{(k)}$), as well as their relative weights ($\phi^{(1)}$ to $\phi^{(j)}$). Unlike *k*-means, though, EM uses soft cluster assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters). Then, during the maximisation step, each cluster is updated using all the instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster. These probabilities are called the responsibilities of the clusters for the instances. During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.

> Unfortunately, just like *k*-means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to 10. By default `n_init` is set to 1.

We can check whether or not the algorithm converged and how many iterations it took:

C.R. 56
```python
print(gm.converged_)
```
python

C.R. 57
```
True
```
text

C.R. 58
```python
print(gm.n_iter_)
```
python

C.R. 59
```
4
```
text

Now that we have an estimate of the location, size, shape, orientation, and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). Just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

C.R. 60
```python
print(gm.predict(X))
```

C.R. 61
```text
[2 2 0 ... 1 1 1]
```

C.R. 62
```python
print(gm.predict_proba(X).round(3))
```

C.R. 63
```text
[[0.    0.023 0.977]
 [0.001 0.016 0.983]
 [1.    0.    0.   ]
 ...
 [0.    1.    0.   ]
 [0.    1.    0.   ]
 [0.    1.    0.   ]]
```

A Gaussian mixture model is a generative model, meaning we can sample new instances from it (note that they are ordered by cluster index):

C.R. 64
```python
X_new, y_new = gm.sample(6)
print(X_new)
```

C.R. 65
```text
[[-2.32491052  1.04752548]
 [-1.16654983  1.62795173]
 [ 1.84860618  2.07374016]
 [ 3.98304484  1.49869936]
 [ 3.8163406   0.53038367]
 [ 0.38079484 -0.56239369]]
```

C.R. 66
```python
print(y_new)
```

C.R. 67
```text
[0 0 1 1 1 2]
```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method estimates the log of the Portable Document Format (PDF) at that location. The greater the score, the higher the density:

C.R. 68
```python
print(gm.score_samples(X).round(2))
```

C.R. 69
```text
[-2.61 -3.57 -3.33 ... -3.51 -4.4  -3.81]
```

If we compute the exponential of these scores, we get the value of the PDF at the location of the given instances. These are not probabilities, but probability densities: they can take on any positive value, not just a value between 0 and 1. To estimate the probability that an instance will fall within

a particular region, we would have to integrate the PDF over that region (if we do so over the entire space of possible instance locations, the result will be 1). **Fig.** 5.17 shows the cluster means, the decision boundaries (dashed lines), and the density contours of this model.

The algorithm clearly found an excellent solution. Of course, we made its task easy by generating the data using a set of 2D Gaussian distributions[22]. We also gave the algorithm the correct number of clusters. When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution. We might need to reduce the difficulty of the task by limiting the number of parameters that the algorithm has to learn. One way to do this is to limit the range of shapes and orientations that the clusters can have. This can be achieved by imposing constraints on the covariance matrices. To do this, set the `covariance_type` hyperparameter to one of the following values:

[22] unfortunately, real-life data is not always so Gaussian and low-dimensional

spherical All clusters must be spherical, but they can have different diameters (i.e., different variances).

diag Clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal).

tied All clusters must have the same ellipsoidal shape, size, and orientation (i.e., all clusters share the same covariance matrix).

By default, `covariance_type` is equal to "full", which means that each cluster can take on any shape, size, and orientation (it has its own unconstrained covariance matrix). **Fig.** 5.18 plots the solutions found by the EM algorithm when `covariance_type` is set to `"tied"` or `"spherical"`.

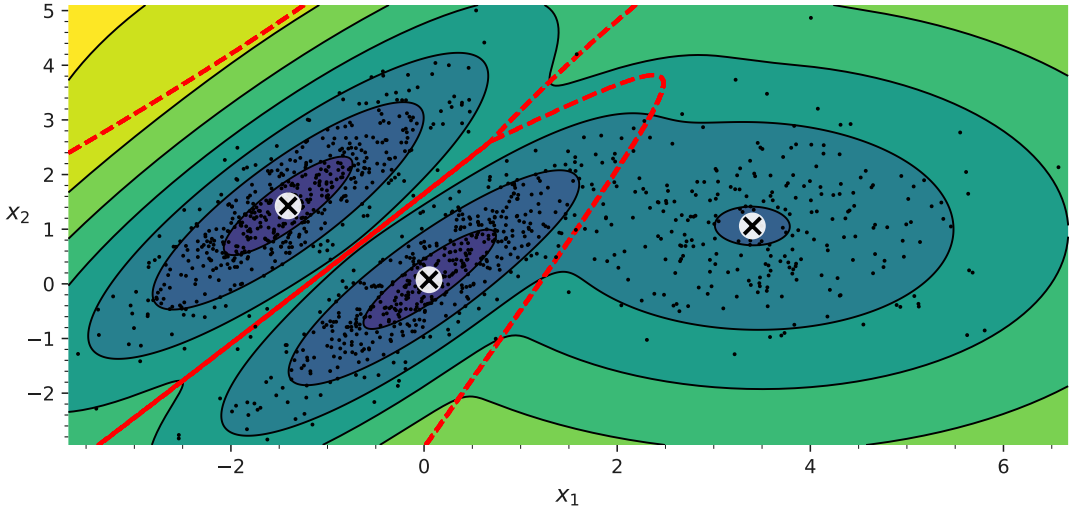


Figure 5.17: Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model.

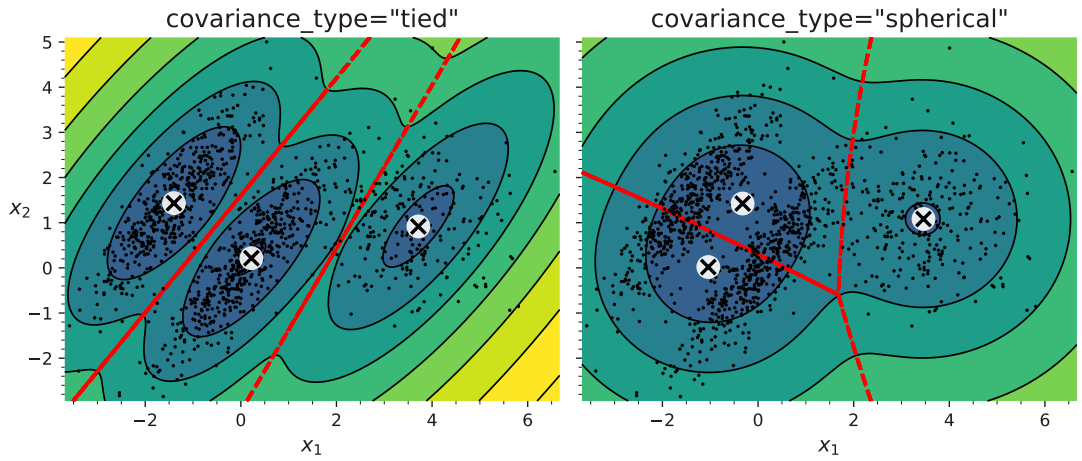


Figure 5.18: Gaussian mixtures for tied clusters (left) and spherical clusters (right)

Information: Computational Complexity

The computational complexity of training a `GaussianMixture` model depends on the number of instances m, the number of dimensions n, the number of clusters k, and the constraints on the covariance matrices. If `covariance_type` is `"spherical"` or `"diag"`, it is O(kmn), assuming the data has a clustering structure. If `covariance_type` is "tied" or "full", it is O(kmn2 + kn3), so it will not scale to large numbers of features.

Gaussian mixture models can also be used for anomaly detection. We'll see how in the next section.

5.3.1 Using Gaussian Mixtures for Anomaly Detection

Using a Gaussian mixture model for anomaly detection is quite simple.

> Any instance located in a low-density region can be considered an anomaly.

We must define what density threshold we want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well known. Say it is equal to 2%. We then set the density threshold to be the value that results in having 2% of the instances located in areas below that threshold density. If we notice that we get too many false positives (i.e., perfectly good products that are flagged as defective), we can lower the threshold. Conversely, if we have too many false negatives (i.e., defective products that the system does not flag as defective), we can increase the threshold. This is the usual precision/recall trade-off. Here is how we would identify the outliers using the fourth percentile lowest density as the threshold (i.e., approximately 4% of the instances will be flagged as anomalies):

C.R. 70
python
```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 2)
anomalies = X[densities < density_threshold]
```

Fig. 5.19 represents these anomalies as stars. A closely related task is novelty detection. It differs from anomaly detection in that the algorithm is assumed to be trained on a **clean** dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption. Indeed, outlier detection is often used to clean up a dataset.

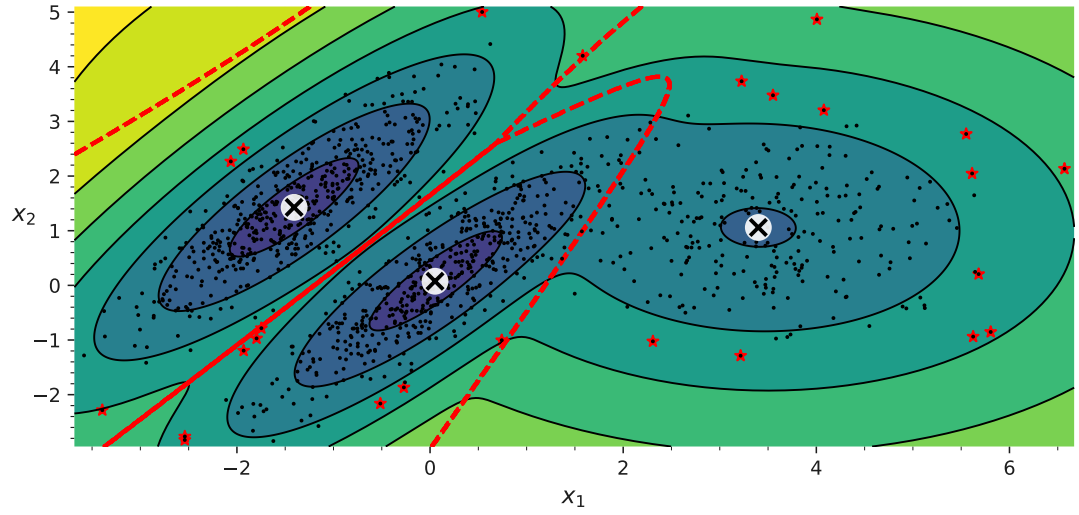


Figure 5.19: Anomaly detection using a Gaussian mixture model.

Information: Working with Outliers

Gaussian mixture models try to fit all the data, including the outliers; if we have too many of them this will bias the model's view of "normality", and some outliers may wrongly be considered as normal. If this happens, we can try to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned-up dataset. Another approach is to use robust covariance estimation methods.

Just like *k*-means, the `GaussianMixture` algorithm requires we to specify the number of clusters. So how can we find that number?

5.3.2 Selecting the Number of Clusters

With *k*-means, we can use the inertia or the silhouette score to select the required number of clusters. But with Gaussian mixtures, it is not possible to use these metrics as they are **NOT** reliable when the clusters are not spherical or have different sizes. Instead, we can try to find the model which minimises a theoretical information criterion, such as the Bayesian Information Criterion (BIC) or

the Akaike Information Criterion (AIC), defined as:

$$\text{BIC} = \log(m)\, p - 2\log\left(\hat{L}\right)$$
$$\text{AIC} = 2p - 2\log\left(\hat{L}\right)$$

where m is the number of instances, p is the number of parameters learned by the model and, $\hat{L}$ is the maximised value of the likelihood function of the model. Both BIC and AIC penalise models with more parameters to learn[23] and reward models which fit the data well. Both methods often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler[24] than the one selected by the AIC, but tends to not fit the data quite as well.[25]

[23] e.g., more clusters

[24] fewer parameters.

[25] this is especially true for larger datasets.

Information: The Likelihood Function

The terms **probability** and **likelihood** are often used interchangeably in everyday language, but have very different meanings in statistics and therefore is worth looking at it in detail.

Given a statistical model with some parameters θ, the word "probability" is used to describe how plausible a future outcome x is,[26] whereas the word "likelihood" is used to describe how **plausible** a particular set of parameter values θ are, after the outcome x is known.

[26] knowing the parameter values θ

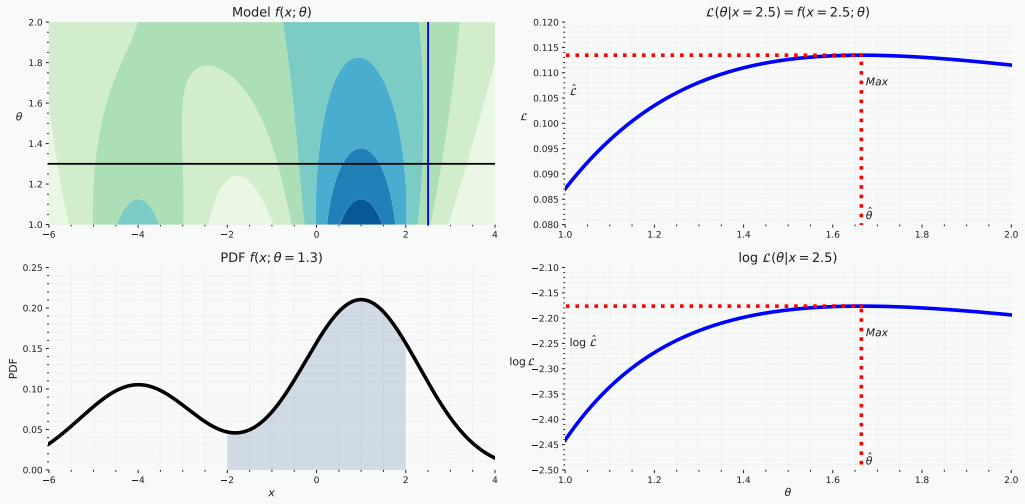


Figure 5.20: A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right).

As an example, consider a 1D mixture model of two Gaussian distributions centered at (-4,1). For simplicity, this toy model has a single parameter (θ) which controls the standard deviations of both distributions. The top-left contour plot in **Fig.** 5.20 shows the entire model $g(x;\theta)$ as a function of both x and θ. To estimate the probability distribution of a future outcome x, we need to set the model parameter θ.

For example, setting θ to 1.3 (the horizontal line), we get the probability density function $f(x;\theta=1.3)$ shown in the lower-left plot. Say we want to estimate the probability that x will fall between -2 and +2. We must calculate the integral of the PDF on this range.[27] But what if we don't know θ, and instead if we have observed a single instance x=2.5 (the vertical line in the upper-left plot)? In this

[27] i.e., the surface of the shaded region.

case, we get the likelihood function $L(\theta|x = 2.5) = f(x = 2.5; \theta)$, represented in the upper-right plot.

In short, the PDF is a function of x (with θ fixed), while the likelihood function is a function of θ (with x fixed). It is important to understand that the likelihood function is not a probability distribution: if we integrate a probability distribution over all possible values of x, we always get 1, but if we integrate the likelihood function over all possible values of θ the result can be any positive value. Given a dataset $\boldsymbol{X}$, a common task is to try to estimate the most likely values for the model parameters. To do this, we must find the values that maximize the likelihood function, given X. In this example, if we have observed a single instance $x = 2.5$, the Maximum Likelihood Estimate (MLE) of θ is $\hat{\theta} = 1.5$. If a prior probability distribution g over θ exists, it is possible to take it into account by maximising $\mathcal{L}\left(\theta|x\right) g\left(\theta\right)$ rather than just maximising $\mathcal{L}\left(\theta|x\right)$. This is called Maximum a-Posteriori (MAP) estimation. Since MAP constrains the parameter values, we can think of it as a regularised version of MLE. Notice that maximising the likelihood function is equivalent to maximising its logarithm (represented in the lower-right plot in **Fig.** 5.20). Indeed, the logarithm is a strictly increasing function, so if θ maximises the log-likelihood, it also maximises the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if we observed several independent instances x(1) to x(m), we would need to find the value of θ that maximises the product of the individual likelihood functions. But it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums: log(ab) = log(a) + log(b). Once we have estimated $\hat{\theta}$, the value of θ that maximises the likelihood function, then we are ready to compute $\boldsymbol{L} = \mathcal{L}\left(\hat{\theta}, \boldsymbol{X}\right)$, which is the value used to compute the AIC and BIC; we can think of it as a measure of how well the model fits the data.

To calculate the values for BIC and AIC, call the `bic()` and `aic()` methods:

Figure 9-20 shows the BIC for different numbers of clusters k. As we can see, both the BIC and the AIC are lowest when k=3, so it is most likely the best choice.

5.3.3 Bayesian Gaussian Mixture Models

Rather than manually searching for the optimal number of clusters and slow down the process, we can use the `BayesianGaussianMixture` class, which is capable of giving weights equal (or close) to zero to unnecessary clusters. Set the number of clusters `n_components` to a value that we have good reason to believe is greater than the optimal number of clusters,[28] and the algorithm will eliminate the unnecessary clusters automatically.

[28] this assumes some minimal knowledge about the problem at hand.

For example, let's set the number of clusters to 10 and see what happens:

Excellent. The algorithm automatically detected only three clusters are needed, and the resulting clusters are almost identical to the ones in **Fig.** 5.19.

An important point to mention about Gaussian mixture models:

although they work great on clusters with ellipsoidal shapes, they don't do so well with clusters of very different shapes.

For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see Figure 9-21).

The algorithm desperately searched for ellipsoids, so it found eight different clusters instead of two. The density estimation is not too bad, so this model could perhaps be used for anomaly detection, but it failed to identify the two moons.

5.3.4 Other Algorithms for Anomaly and Novelty Detection

Of course there are other algorithms as well with `sklearn` implementing other algorithms dedicated to anomaly detection or novelty detection:

Fast-MCD Stands for **minimum covariance determinant**. Implemented by the `EllipticEnvelope` class, this algorithm is useful for outlier detection, in particular to clean up a dataset. It assumes the normal instances (inliers) are generated from a single Gaussian distribution.[29] It also assumes the dataset is contaminated with outliers which were not generated from this Gaussian distribution.

[29] Which is not a mixture.

MCD is a method for estimating the mean and covariance matrix in a way that tries to minimize the influence of anomalies. When the algorithm estimates the parameters of the Gaussian distribution,[30] it is careful to ignore the instances that are most likely outliers.

[30] i.e., the shape of the elliptic envelope around the inliers.

This technique gives a better estimation of the elliptic envelope and thus makes the algorithm better at identifying the outliers.

Isolation Forest An efficient algorithm for outlier detection, especially in high-dimensional datasets. The algorithm builds a random forest in which each decision tree is grown randomly. At each node, it picks a feature randomly, then it picks a random threshold value[31] to split the dataset in two. The dataset gradually gets chopped into pieces this way, until all instances end up isolated from the other instances. Anomalies are usually far from other instances, so on average[32] they tend to get isolated in fewer steps than normal instances [31].

[31] between the min and max values

[32] across all the decision trees

Local Outlier Factor Used in outlier detection as it compares the density of instances around a given instance to the density around its neighbors. An anomaly is often more isolated than its $k$nearest neighbours.

One-Class SVM Suited for novelty detection. Recall that a kernelized SVM classifier separates two classes by first (implicitly) mapping all the instances to a high-dimensional space, then separating

the two classes using a linear SVM classifier within this high-dimensional space. Since we just have one class of instances, the one-class SVM algorithm instead tries to separate the instances in high-dimensional space from the origin. In the original space, this will correspond to finding a small region that encompasses all the instances. If a new instance does not fall within this region, it is an anomaly. There are a few hyperparameters to tweak: the usual ones for a kernelized SVM, plus a margin hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel when it is in fact normal. It works great, especially with high-dimensional datasets, but like all SVMs it does not scale to large datasets.

PCA and other dimensionality reduction techniques with an `inverse_transform()` method If we compare the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger. This is a simple and often quite efficient anomaly detection approach.

Part II

Neural Networks

There must be a trick to the train of thought, a recursive formula. A group of neurons starts working automatically, sometimes without external impulse. It is a kind of iterative process with a growing pattern. It wanders about in the brain, and the way it happens must depend on the memory of similar patterns.

(Stanislaw M. Ulam, Adventures of a Mathematician)

Chapter 6

Introduction to Artificial Neural Networks

Table of Contents

6.1 Introduction

It is quite apparent that life imitates life and engineers are inspired by nature. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine.

This is the logic that sparked ANNs, ML models inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don't have to flap their wings to fly. Similarly, ANNs have gradually become quite different from their biological cousins. Some

Figure 6.1: Nature always is a great source of inspiration for good design. For example, the beak of a bird is aerodynamically efficient and was used in designing the Bullet train [32].The field is of emulating models, systems, and elements of nature for the purpose of solving complex human problems is called bio mimetics [33].

Figure 6.2: The prolific advancements of computers and neural networks have allowed us to tackle problems once deemed impossible. A game of GO requires uncountable amount of moves, yet using ML it was possible to create a software capable of beating the world champion.

researchers even argue that we should drop the biological analogy altogether such as calling them units rather than neurons [34], as some consider this naming to decrease the amount of creativity we can give to the topic.

ANNs are at the very core of **deep learning**. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex ML tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of Go (DeepMind's AlphaGo [35]).

We will treat this chapter as a formal introduction to ANN, starting with a tour of the very first ANN architectures and leading up to multilayer perceptrons, which are heavily used today.

In the second part, we will look at how to implement neural networks using TensorFlow's Keras API. This is a beautifully designed and simple high-level API for building, training, evaluating, and running neural networks. While it may look simple at first glance, it is expressive and flexible enough to let you build a wide variety of neural network architectures.

> For most of your use cases, using `keras` will be enough.

6.2 From Biology to Silicon: Artificial Neurons

While it may seem they are the cutting edge in ML, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their landmark paper *A Logical Calculus of Ideas Immanent in Nervous Activity*, They presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic. This was the first artificial neural network architecture [36].

Since then many other architectures have been invented. The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. This is also known as the 1st AI winter [37]. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in connectionism, the study of neural networks. But progress was slow, and by the 1990s other powerful ML techniques had been invented, such as SVM. These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold and this is known as the 2nd AI winter.

We are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

There is now a **huge quantity of data available** to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems. One of the major turning points of ANN was the fundamental question of:

Is our understanding of the model at fault or is it merely the lack of data to train?

The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to **Moore's law**, but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions which have become the norm to train ML instead of CPUs.

Information: Moore's Law

the number of components in integrated circuits has doubled about every 2 years over the last 50 years.

In addition to previous additions, cloud platforms have made this power accessible to everyone. The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.

Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in

local optima [38], but it turns out that this is not a big problem in practice, especially for larger neural networks: the local optima often perform almost as well as the global optimum.

ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

6.2.1 Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron. It is an unusual-looking cell mostly found in animal brains.

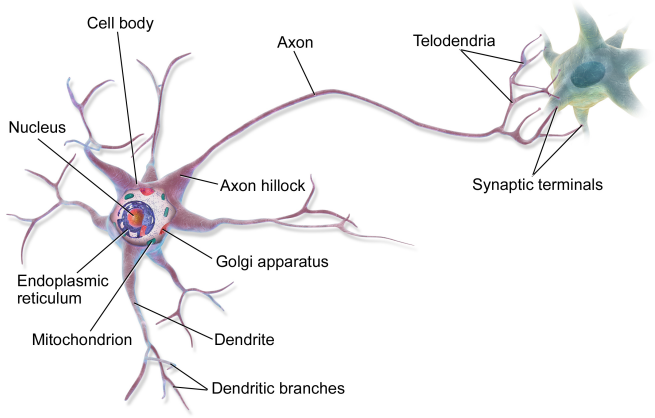


Figure 6.3: A neuron or nerve cell is an excitable cell that fires electric signals called action potentials across a neural network in the nervous system. Neurons communicate with other cells via synapses, which are specialized connections that commonly use minute amounts of chemical neurotransmitters to pass the electric signal from the presynaptic neuron to the target cell through the synaptic gap [39].

It's composed of a cell body containing the nucleus and most of the cell's complex components, many branching extensions called dendrites, plus one very long extension called the axon.The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer.

Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called synaptic terminals (or simply synapses), which are connected to the dendrites or cell bodies of other neurons. Biological neurons produce short electrical impulses called action potentials (APs, or just signals), which travel along the axons and make the synapses release chemical signals called neurotransmitters. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).

Therefore, individual biological neurons seem to behave in a simple way, but they're organised in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural

networks (BNNs) is the subject of active research, but some parts of the brain have been mapped [40]. These efforts show that neurons are often organised in consecutive layers, especially in the cerebral cortex.[1]

[1] the outer layer of the brain

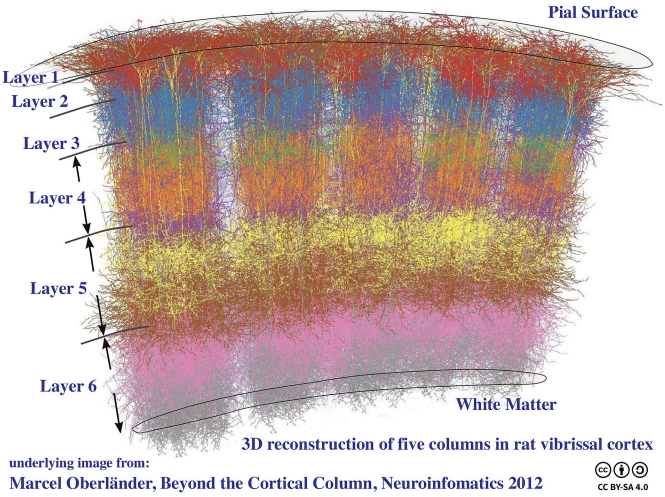


Figure 6.4: A cortical column is a group of neurons forming a cylindrical structure through the cerebral cortex of the brain perpendicular to the cortical surface. The structure was first identified by Vernon Benjamin Mountcastle in 1957. He later identified minicolumns as the basic units of the neocortex which were arranged into columns. Each contains the same types of neurons, connectivity, and firing properties. Columns are also called hypercolumn, macrocolumn, functional column or sometimes cortical module.

6.2.2 Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an **artificial neuron**: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. In their paper, McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that can compute any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computationsm, assuming that a neuron is activated when at least two of its input connections are active. Let's see what these networks do:

- The first network on the left is the identity function: if neuron **A** is activated, then neuron **C** gets activated as well (since it receives two input signals from neuron **A**); but if neuron **A** is off, then neuron **C** is off as well.
- The second network performs a logical **AND**: neuron **C** is activated only when both neurons **A** and **B** are activated (a single input signal is not enough to activate neuron **C**).
- The third network performs a logical **OR**: neuron **C** gets activated if either neuron **A** or neuron **B** is activated (or both).

- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can imagine how these networks can be combined to compute complex logical expressions.

6.2.3 The Perceptron

The perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt [41]. It is based on a slightly different artificial neuron called a Threshold Logic Unit (TLU), or sometimes a Linear Threshold Unit (LTU) which can be seen in Fig. 6.5. The inputs and output are numbers (this is instead of binary on/off values), and each input connection is associated with a weight. The TLU first computes a linear function of its inputs:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \boldsymbol{x}^\top \boldsymbol{w} + b$$

Then it applies a **step function** to the result:

$$h(x) = \text{step}(z) \qquad \text{where} \qquad z = \boldsymbol{x}^\top \boldsymbol{w}.$$

It is similar to logistic regression, except it uses a step function instead of the logistic function. Just like in logistic regression, the model parameters are the input weights w and the bias term b.

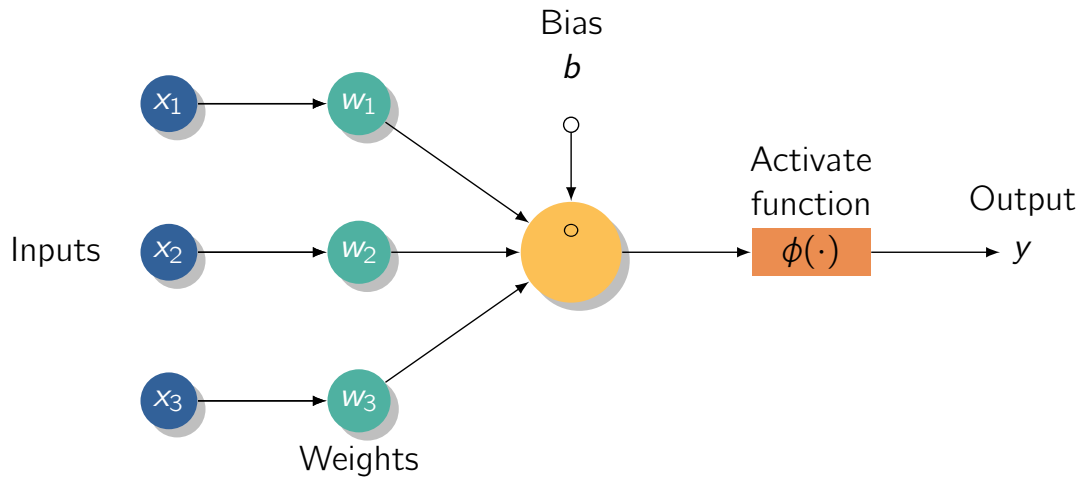


Figure 6.5: Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a certain activation function.

The most common step function used in perceptrons is the **Heaviside step** and sometimes the **sign**

function is used instead [42]. 5

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

A single TLU can be used for simple **linear binary classification**:

> *It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class. They exhibit a similar behaviour to logistic regression or linear SVM classification.*

It is possible, for example, use a single TLU to classify iris flowers [43] (a famous dataset used by statisticians and ML researchers) based on petal length and width. Training such a TLU would require finding the right values for w_1, w_2, b.

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a **fully connected layer**, or a dense layer. The inputs constitute the input layer and since the layer of TLUs produces the final outputs, it is called the output layer.

This perceptron can classify instances simultaneously into three (3) different binary classes, which makes it a **multilabel classifier**. It may also be used for multiclass classification.

Using linear algebra, the following equation can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

$$h_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{X}) = \phi(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b})$$

In this equation:

- $\boldsymbol{X}$ represents the matrix of input features. It has one row per instance and one column per feature.
- $\boldsymbol{W}$ is the weight matrix containing all the connection weights. It has one row per input and one column per neuron.
- $\boldsymbol{b}$ is the bias term containing all the bias terms: one per neuron.
- ϕ is the activation function is called the activation function: when the artificial neurons are TLUs, it is a step function.

Now the question is:

> *How is this perceptron train?*

The original perceptron training algorithm proposed by Rosenblatt was largely inspired by Hebb's rule [44]. In his 1949 book *The Organization of Behaviour*, Donald Hebb suggested that when a

biological neuron triggers another neuron often, the connection between these two neurons grows stronger [45].

Siegrid Löwel later summarized Hebb's idea in the catchy phrase,

> *Cells that fire together, wire together*

This means the connection weight between two neurons tends to increase when they fire simultaneously.

> This rule later became known as Hebb's rule (or Hebbian learning [46])

Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction. The perceptron learning rule **reinforces connections that help reduce the error**.

More specifically, the perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

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

where:

- $w_{i,j}$ is the connection weight between the i^{th} input and the j^{th} neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}^j$ is the output of the j^{th} output neuron for the current training instance.
- y^j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

The decision boundary of each output neuron is **linear**, therefore perceptrons are incapable of learning complex patterns. However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.

> This is called the perceptron convergence theorem.

C.R. 1

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values y = (iris.target == 0) # Iris
↪ setosa
per_clf = Perceptron(random_state=42) per_clf.fit(X, y)
X_new = [[2, 0.5], [3, 1]]
```

```python
y_pred = per_clf.predict(X_new) # predicts True and False for these 2 flowers
```
C.R. 2

For those of you who have taken a **Data Science II** course, you may have noticed that the perceptron learning algorithm strongly resembles *stochastic gradient descent*. In fact, `sklearn`'s `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters:

- `loss="perceptron"`,
- `learning_rate="constant"`,
- `eta0=1` (the learning rate),
- `penalty=None` (no regularization).

In their 1969 monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of perceptrons: in particular, they are incapable of solving some trivial problems (e.g., the exclusive OR (XOR) classification problem).

Information: XOR Problem

A simple logic gate problem which is proven to be unsolvable using a single-layer perceptron.

This is true of any other linear classification model, but researchers had expected much more from perceptrons, and some were so disappointed, they dropped neural networks altogether in favour of higher-level problems such as logic, problem solving, and search. The lack of practical applications also didn't help.

It turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting ANN is called a MLP and a MLP can solve the XOR problem [47].

Perceptrons **DO NOT** output a class probability. This is one reason to prefer logistic regression over perceptrons. Moreover, perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as logistic regression or a linear SVM classifier. However, perceptrons may train a bit faster.

6.2.4 Multilayer Perceptron and Backpropagation

An MLP is composed of one input layer, one or more layers of TLUs called **hidden layers**, and one final layer of TLUs called the output layer. The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers.

The signal flows only in one direction (inputs to outputs), so this architecture is an example of a Feedforward Neural Networks (FNN) [48].

When an ANN contains a deep stack of hidden layers, it is called a Deep Neural Networks (DNN). The field of deep learning studies DNNs, and more generally it is interested in models containing deep stacks of computations [49]. For many years researchers struggled to find a way to train MLPs,

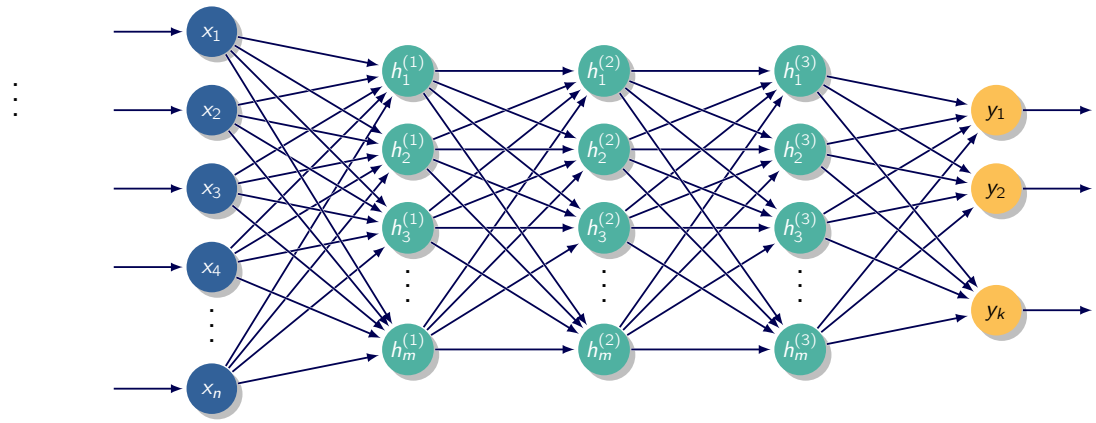


Figure 6.6: Architecture of a Multilayer Perceptron with five inputs, three hidden layer of four neurons, and three output neurons.

without success. In the early 1960s several researchers discussed of using gradient descent to train neural networks. This requires computing the gradients of the model's error with regard to the model parameters and at that time, it wasn't clear at the time how to do this efficiently with such a complex model containing so many parameters.

Then, in 1970, a researcher named *Seppo Linnainmaa* introduced in his master's thesis a technique to compute all the gradients automatically and efficiently. This algorithm is now called **reverse-mode automatic differentiation** [50]. In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network's error with regard to every single model parameter.

In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network's error. These gradients can then be used to perform a gradient descent step. Repeating the process of computing the gradients automatically and taking a gradient descent step, the neural network's error will gradually drop until it eventually reaches a minimum.

This combination of reverse-mode automatic differentiation and gradient descent is now called **backpropagation** [51].

There are various automatic differentiation techniques (i.e., forward and reverse), with each having its own advantages and disadvantages. Reverse-mode autodiff is well suited when the function to differentiate has many variables (e.g., connection weights and biases) and few outputs (e.g., one loss).

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: Linnainmaa's M.Sc thesis was not about neural nets, it was more general. It was several more years before backprop started to be used to train neural networks, but it still wasn't mainstream.

Then, in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper analyzing how backpropagation allowed neural networks to learn useful internal representations [52]. Their results were so impressive that backpropagation was quickly popularized in the field. Today, it is by far the most popular training technique for neural networks.

Let's run through how backpropagation works again in a bit more detail:

1. It handles one mini-batch at a time, and goes through the full training set multiple times. Each pass is called an **epoch**.

2. Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first hidden layer, for every instance in the mini-batch. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer.

3. Next, the algorithm measures the network's output error. This means, it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error.

4. It then computes how much each output bias and each connection to the output layer contributed to the error. This is done analytically by applying the chain rule, which makes this step fast and precise.

5. The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights and biases in the network by propagating the error gradient backward through the network.

6. Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

> Initialize all the hidden layers' connection weights randomly, or training will fail.

For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and therefore backpropagation will affect them in exactly the same way, so they will remain identical.

In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you **break the symmetry** and allow back-propagation to train a diverse team of neurons.

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error, which is the gradient descent step.

For back-propagation to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture by replacing the step function with the logistic function:

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

Which is also called the **sigmoid function**. This was an important improvement as step function contains only flat segments, so there is no gradient to work with, while the sigmoid function has a well-defined nonzero derivative everywhere. In fact, the backpropagation algorithm works well with many other activation functions, not just the sigmoid function.

Here are two (2) other popular choices:

Exercise 6.1: Hyperbolic tangent function

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

Similar sigmoid function, this activation function is also S-shaped, continuous, and differentiable, but its output value ranges from -1 to 1, instead of 0 to 1 like the sigmoid function.
This bigger range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

Exercise 6.2: The rectified linear unit function

$$ReLU(z) = max(0, z)$$

It is continuous but unfortunately not differentiable at $z = 0$ as the slope changes abruptly, which can make gradient descent bounce around, and its derivative is 0 for $z < 0$.
In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.
Importantly, the fact that it does not have a maximum output value helps reduce some issues during gradient descent.

You might wonder what is the point of an activation function, let alone whether it is linear or not? Chaining several linear transformations, gives you only linear transformation. For example:

$$f(x) = 2x + 3 \quad \text{and} \quad g(x) = 5x - 1 \quad \rightarrow \quad f\left(g(x)\right) = 2(5x - 1) + 3 = 10x + 1$$

You don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that.

> A large enough DNN with nonlinear activations can theoretically approximate any continuous function.

6.2.5 Regression MLPs

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), you just need a single output neuron:

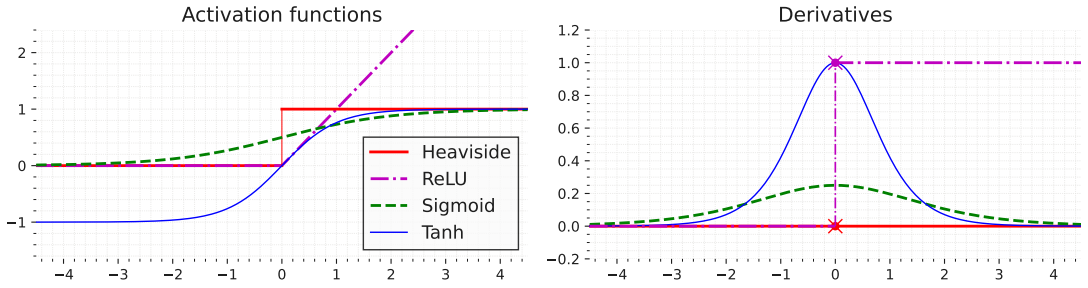


Figure 6.7: The activation function of a node in an ANN is a function which calculates the output of the node based on its individual inputs and their weights. Nontrivial problems can be solved using only a few nodes if the activation function is nonlinear [53]. Modern activation functions include the smooth version of the ReLU, the GELU, which was used in the 2018 BERT model [54], the logistic (sigmoid) function used in the 2012 speech recognition model developed by Hinton et al [55], the ReLU used in the 2012 AlexNet computer vision model [56] and in the 2015 ResNet model.

its output is the predicted value

For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. As an example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two (2) output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object.

So, in the end you end up with four (4) output neurons.

`sklearn` includes an `MLPRegressor` class, so let's use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset.

For simplicity, we will use `sklearn`'s `fetch_california_housing()` function to load the data instead of downloading from a sketchy website.

The following code starts by fetching and splitting the dataset, then it creates a pipeline to standardise the input features before sending them to the `MLPRegressor`. This is very important for neural networks as they are trained using gradient descent, and gradient descent does not converge very well when the features have very different scales.

Finally, the code trains the model and evaluates its validation error. The model uses the ReLU activation function in the hidden layers, and it uses a variant of gradient descent called Adam to minimize the mean squared error, with a little bit of ℓ_2 regularisation:

C.R. 3
python
```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
```

C.R. 4 python
```
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)
```

We get a validation RMSE of about 0.505, which is comparable to what you would get with a random forest classifier.

> This MLP does not use any activation function for the output layer, so it's free to output any value it wants.

This is generally fine, but if you want to guarantee that the output will always be positive, then you should use the ReLU activation function in the output layer, or the softplus activation function, which is a smooth variant of ReLU: softplus(z) = log(1 + exp(z)).

Softplus is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and -1 to 1 for tanh.

Sadly, the `MLPRegressor` class does not support activation functions in the output layer.

> Building and training a standard MLP with `sklearn` is very convenient, but features are limited. This is why we will switch to Keras in the second part of this chapter.

The `MLPRegressor` class uses the mean squared error, which is usually what you want for regression, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you may want to use the Huber loss, which is a combination of both. It is quadratic when the error is smaller than a threshold δ (typically 1) but linear when the error is larger than δ. The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error. However, `MLPRegressor` only supports the MSE.

6.2.6 Classification MLPs

MLPs can also be used for **classification** tasks. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class.

> The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks. For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email.

In this case, you would need two output neurons, both using the sigmoid activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see Figure 10-9). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1, since the classes are exclusive. As you saw in Chapter 3, this is called multiclass classification.

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (or x-entropy or log loss for short, see Chapter 4) is generally a good choice.

`sklearn` has an `MLPClassifier` class in the `sklearn.neural_network` package. It is almost identical to the `MLPRegressor` class, except that it minimizes the cross entropy rather than the MSE. Give it a try now, for example on the iris dataset. It's almost a linear task, so a single layer with 5 to 10 neurons should suffice (make sure to scale the features).

6.3 Implementing MLPs with Keras

Keras is TensorFlow's high-level deep learning API: it allows you to build, train, evaluate, and execute all sorts of neural networks. The original Keras 12 library was developed by Francois Chollet as part of a research project and was released as a standalone open source project in March 2015. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design.

Information: Application

Keras used to support multiple backends, including TensorFlow, PlaidML, Theano, and Microsoft Cognitive Toolkit (CNTK) (the last two are sadly deprecated), but since version 2.4, Keras is TensorFlow-only. Similarly, TensorFlow used to include multiple high-level APIs, but Keras was officially chosen as its preferred high-level API when TensorFlow 2 came out. Installing TensorFlow will automatically install Keras as well, and Keras will not work without TensorFlow installed. In short, Keras and TensorFlow fell in love and got married. Other popular deep learning libraries include PyTorch by Facebook and JAX by Google.13

6.3.1 Building an Image Classifier Using Sequential API

Before we do anything, we need to load a dataset. We will use Fashion MNIST. There are 70,000 grayscale images of 28 Œ 28 pixels each, with 10 classes where images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly challenging.

Using Keras to load the dataset

`keras` provides utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, and a few more.

Let's load Fashion MNIST. It's already shuffled and split into a training set (60,000 images) and a test set (10,000 images), but we'll hold out the last 5,000 images from the training set for validation:

C.R. 5

```python
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

TensorFlow is usually imported as `tf`, and the Keras API is available via `tf.keras`.

When loading MNIST or Fashion MNIST using `tf.keras` rather than `sklearn`, an important difference is that every image is represented as a 28-by-28 array rather than a 1D array of size 784 with intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0).

Let's take a look at the shape and data type of the training set:

C.R. 6 python
```
print("The size of the training dataset: ", X_train.shape)
print("The type of the training dataset: ", X_train.dtype)
```

C.R. 7 text
```
The size of the training dataset:  (55000, 28, 28)
The type of the training dataset:  uint8
```

To make it simple, we'll scale the pixel intensities down to the 0-1 range by dividing them by 255.0

This operation also converts the integer values to floats.

C.R. 8 python
```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

Using Fashion MNIST, we need the list of class names to know what we are dealing with:

C.R. 9 python
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

For example, the first image in the training set represents an ankle boot: and below we can see

Figure 6.8: An example of a data within the Fashion MNIST.

some examples of the Fashion MNIST dataset.

6.3.2 Creating the model using the sequential API

It is time to build the neural network. Here is a classification MLP with two (2) hidden layers:

Figure 6.9: A random collection of dataset, making the Fashion MNIST.

C.R. 10

```python
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Let's try to understand the code:

1. Set `tf` random seed to make the results reproducible: the random weights of the hidden layers and the output layer will be the same every time you run your code. You could also choose to use the `tf.keras.utils.set_random_seed()` function, which conveniently sets the random seeds for TensorFlow, Python (`random.seed()`), and NumPy (`np.random.seed()`).

2. Next line creates a *Sequential model*. This is the simplest kind of Keras model for neural networks, composed of a single stack of layers connected sequentially. This is called the sequential API.

3. We build the first layer (an Input layer) and add it to the model. We specify the input shape, which doesn't include the batch size, only the shape of the instances. Keras needs to know the shape of the inputs so it can determine the shape of the connection weight matrix of the first hidden layer.

4. We add a Flatten layer. Its role is to convert each input image into a 1D array: for example, if it receives a batch of shape [32, 28, 28], it will reshape it to [32, 784]. In other words, if it receives input data X, it computes `X.reshape(-1, 784)`. This layer doesn't have any parameters; it's just there to do some simple pre-processing.

5. We add a Dense hidden layer with 300 neurons. It will use the ReLU activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the

neurons and their inputs. It also manages a vector of bias terms (one per neuron).

6. We add a second Dense hidden layer with 100 neurons, also using the ReLU activation function.

7. We add a Dense output layer with 10 neurons (one per class), using the softmax activation function because the classes are exclusive.

> Writing the argument `activation="relu"` is equivalent to specifying `activation=tf.keras.activations.relu`. Other activation functions are available in the `tf.keras.activations` package.

Instead of adding the layers one by one as we just did, it's often more convenient to pass a list of layers when creating the Sequential model. You can also drop the Input layer and instead specify the `input_shape` in the first layer:

C.R. 11
```python
tf.keras.backend.clear_session()
tf.random.set_seed(42)

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

The model's `summary()` method displays all the model's layers, including each layer's name, which is automatically generated, its output shape, and its number of parameters.

The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters:

C.R. 12
```python
tf.keras.utils.plot_model(model, imagePath+"mnist-_model.pdf", show_shapes=True)
```

Dense layers often have a lot of parameters. For example, the first hidden layer has 784-by-300 connection weights, with 300 bias terms, which adds up to 235,500 parameters.

This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of **over-fitting**, especially when you do not have a lot of training data.

Each layer in a model must have a unique name (e.g., `dense_2`). You can set the layer names explicitly using the constructor's name argument, but generally it's simpler to let Keras name the layers automatically, as we just did. Keras takes the layer's class name and converts it to snake case (i.e., a layer from the `MyCoolLayer` class is named `my_cool_layer` by default). Keras also ensures that the name is **globally unique**, even across models, by appending an index if needed, as in `dense_2`.

This naming scheme makes it possible to merge models easily without getting name conflicts.

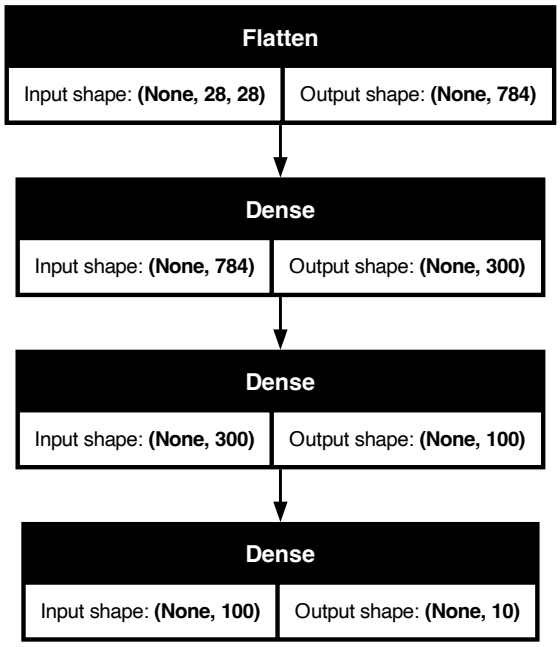


Figure 6.10: The plot of the neural network, showcasing its layers.

> All global state managed by Keras is stored in a Keras session, which you can clear using `tf.keras.backend.clear_session()`.

You can easily get a model's list of layers using the layers attribute, or use the `get_layer()` method to access a layer by name:

C.R. 13
```python
print(model.layers)
```

C.R. 14
```text
[<Flatten name=flatten, built=True>,
<Dense name=dense, built=True>,
<Dense name=dense_1, built=True>,
<Dense name=dense_2, built=True>]
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods.

For a Dense layer, this includes both the connection weights and the bias terms:

C.R. 15
```python
hidden1 = model.layers[1]
weights, biases = hidden1.get_weights()
print(weights)
```

C.R. 16 text

```
[[-0.05415904  0.00010975 -0.00299759 ...  0.05136904  0.0740822
   0.06472497]
 [ 0.05510217 -0.01353022 -0.00363479 ...  0.07100512 -0.04926914
  -0.02905609]
 [-0.07024231  0.02524897 -0.04784295 ... -0.0521326   0.05084455
  -0.06636713]
 ...
 [ 0.0067075  -0.00256791 -0.064556   ...  0.05266081  0.03520959
  -0.02309504]
 [ 0.05826265 -0.0361187  -0.04228947 ...  0.05612285 -0.03179397
   0.06843598]
 [ 0.06636336 -0.00123435 -0.00247347 ...  0.01809192  0.03434542
   0.00700523]]
```

Notice that the Dense layer initialized the connection weights randomly.

> This is needed to break symmetry.

The biases were initialized to zeros, which is fine.

C.R. 17 python

```python
print(biases)
```

C.R. 18 text

```
[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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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.]
```

If you want to use a different initialization method, you can set `kernel_initializer` or `bias_initializer` when creating the layer.

Information: Weight Matrix Shape

The shape of the weight matrix depends on the number of inputs, which is why we specified the `input_shape` when creating the model. If you do not specify the input shape, it's OK: Keras will simply wait until it knows the input shape before it actually builds the model parameters. This will happen either when you feed it some data (e.g., during training), or when you call its build() method. Until the model parameters are built, you will not be able to do certain things, such as display the model summary or save the model. So, if you know the input shape when creating the model, it is best to specify it.

Model Compiling

After a model is created, we need to call its `compile()` method to specify the loss function and the optimizer to use, or you can specify a list of extra metrics to compute during training and evaluation:

C.R. 19
```python
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Before continuing, we need to explain what is going on here.

We use the `sparse_categorical_crossentropy` loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are **exclusive**.

- If we had one target probability per class for each instance (such as one-hot vectors, e.g., [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.] to represent class 3), then we would need to use the `categorical_crossentropy` loss instead.
- If we were doing binary classification or multilabel binary classification, then we would use the sigmoid activation function in the output layer instead of the softmax activation function, and we would use the `binary_crossentropy` loss.

Regarding the optimizer, `sgd` means that we will train the model using stochastic gradient descent. Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff plus gradient descent).

Finally, as this is a classifier, it's useful to measure its accuracy during training and evaluation, which is why we set `metrics=["accuracy"]`.

Training and Evaluating Models

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

C.R. 20
```python
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution).

We also pass a validation set which is optional. Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs.

> If the performance on the training set is much better than on the validation set, the model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set.

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of mini-batches processed so far on the left side of the progress bar.

The batch size is 32 by default, and since the training set has 55,000 images, the model goes through 1,719 batches per epoch: 1,718 of size 32, and 1 of size 24.

After the progress bar, you can see the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set and notice that the training loss went down, which is a good sign, and the validation accuracy reached 88.94% after 30 epochs.

That's slightly below the training accuracy, so there is a little bit of overfitting going on, but not a huge amount.

If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, to give a larger weight to underrepresented classes and a lower weight to overrepresented classes.

These weights would be used by Keras when computing the loss. If you need per-instance weights, set the `sample_weight` argument. If both `class_weight` and `sample_weight` are provided, then Keras multiplies them. Per-instance weights could be useful, for example, if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former.

You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the `validation_data` tuple. The `fit()` method returns a History object containing the training parameters (history.params), the list of epochs it went through (history.epoch), and most importantly a dictionary (history.history) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any).

C.R. 21 python
```python
print(history.params)
print(history.epoch)
```

C.R. 22 text
```
{'verbose': 'auto', 'epochs': 30, 'steps': 1719}
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
↪  26, 27, 28, 29]
```

If you use this dictionary to create a Pandas DataFrame and call its `plot()` method, you get the learning curves shown in Fig. 6.11.

You can see that both the training accuracy and the validation accuracy steadily increase during

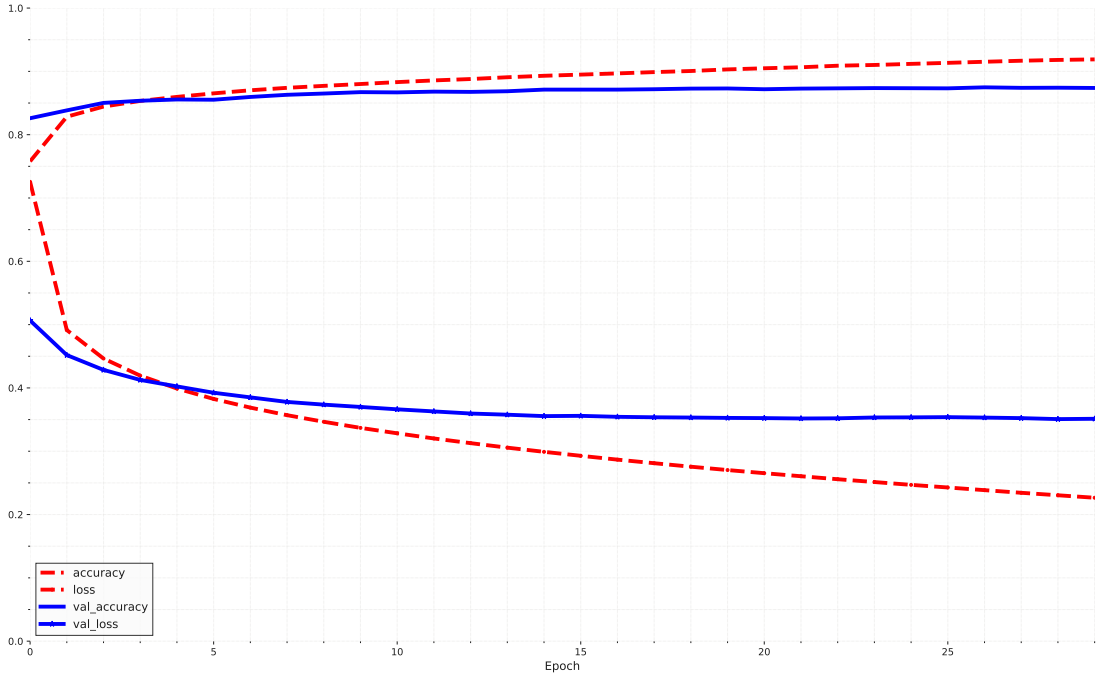


Figure 6.11: Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

training, while the training loss and the validation loss decrease.

This is good.

The validation curves are relatively close to each other at first, but they get further apart over time, which shows that there's a little bit of overfitting. In this particular case, the model looks like it performed better on the validation set than on the training set at the beginning of training, but that's not actually the case.

The validation error is computed at the end of each epoch, while the training error is computed using a running mean during each epoch, so the training curve should be shifted by half an epoch to the left.

If you do that, you will see that the training and validation curves overlap almost perfectly at the beginning of training. The training set performance ends up beating the validation performance, as is generally the case when you train for long enough.

You can tell that the model has not quite converged yet, as the validation loss is still going down, so it would be better to continue training. This is as simple as calling the `fit()` method again, as Keras just continues training where it left off: you should be able to reach about 89.8% validation accuracy, while the training accuracy will continue to rise up to 100%.

> This is not always the case.

If you are not satisfied with the performance of your model, it is a good idea to back and tune the hyperparameters.

1. First check the learning rate (η).
2. If that doesn't help, try another optimizer, and always retune the learning rate after changing any hyperparameter,
3. If the performance is still not great, try tuning model hyperparameters such as the number of layers, the number of neurons per layer, and the types of activation functions to use for each hidden layer.

You can also try tuning other hyperparameters, such as the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32).

Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method.

> It also supports several other arguments, such as `batch_size` and `sample_weight`.

It is common to get slightly lower performance on the test set than on the validation set, as hyperparameters are **tuned on the validation set**, not the test set. However, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck.

> Resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using Model to Make Predictions

It is time to use the model's `predict()` method to make predictions on new instances. As we don't have actual new instances, we'll just use the first three (3) instances of the test set:

C.R. 23

```python
X_new = X_test[:3]
y_proba = model.predict(X_new)
print(y_proba.round(2))
```

C.R. 24

```text
[[0.   0.   0.   0.   0.   0.12 0.   0.01 0.   0.87]
 [0.   0.   1.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]]
```

For each instance the model estimates one probability per class, from class 0 to class 9. This is similar to the output of the `predict_proba()` method in `sklearn` classifiers.

For example, for the first image it estimates that the probability of class 9 (ankle boot) is 87%, the probability of class 7 (sneaker) is 1%, the probability of class 5 (sandal) is 12%, and the probabilities of the other classes are negligible.

In other words, it is highly confident that the first image is footwear, most likely ankle boots but possibly sneakers or sandals. If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `argmax()` method to get the highest probability class index for each instance:

C.R. 25 python
```
y_pred = y_proba.argmax(axis=-1)
print(y_pred)
```

C.R. 26 text
```
[9 2 1]
```

Here, the classifier actually classified all three images correctly, where these images are shown in Fig. 6.12.

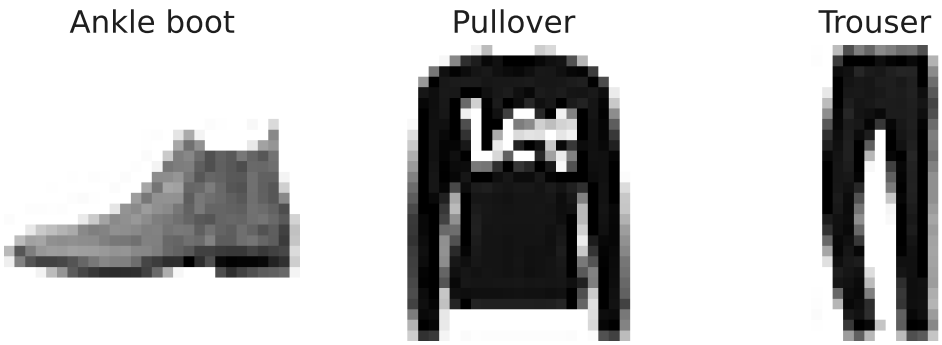


Figure 6.12: Correctly classified Fashion MNIST images.

6.3.3 Building a Regression MLP Using the Sequential API

Instead of classifying categories, lets try to estimate a value. For this application, we need a different dataset. Let's switch back to the California housing problem and tackle it using the same MLP as earlier, with 3 hidden layers composed of 50 neurons each, but this time building it with `tf.keras`.

Using the sequential API to build, train, evaluate, and use a regression MLP is quite similar to what we did for classification. The main differences in the following code example are the fact that the output layer has a single neuron (since we only want to predict a single value) and it uses no activation function, the loss function is the mean squared error, the metric is the RMSE, and we're using an Adam optimizer like `sklearn`s `MLPRegressor` did.

In addition, in this example we don't need a Flatten layer, and instead we're using a Normalization

layer as the first layer: it does the same thing as `sklearns` `StandardScaler`, but it must be fitted to the training data using its `adapt()` method before you call the model's `fit()` method.

Let's look at the code:

C.R. 27 python
```
housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

C.R. 28 python
```
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

As you can see, the sequential API is quite clean and straightforward. However, although Sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the functional API.

Chapter 7

Computer Vision using Convolutional Neural Networks

Table of Contents

7.1 Introduction

It wasn't until recently computers were able to reliably perform seemingly easy tasks such as detecting a cat[1] in a picture or recognising spoken words.

[1] As it is known, internet was invented for sharing cat pictures, therefore their detection is paramount.

Why are these tasks so effortless to us humans?

The answer lies in the fact that perception largely takes place outside the realm of our control, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already imbued with high-level features. For example:

> When we look at a picture of a cute puppy, we cannot choose not to see the puppy, not to notice its cuteness. Nor can we explain how we recognize a cute puppy; it's just obvious to you.

Therefore, we cannot trust our subjective experience.

Perception is **NOT** a trivial task, and to understand it we must look at how our sensory modules work. CNNs emerged from the study of the brain's visual cortex, and they have been used in computer image recognition since the 1980s [57].

Over the last 10 years, thanks to the increase in computational power, the amount of available training data, and a much better methods developed for training deep nets, CNNs have managed to achieve impressive performance on some complex visual tasks. Some examples of their application include:

- **Search services:** Such as connecting users in the web to the software they need to find [58], [59],
- **Self-driving cars:** Such as detecting the colours on a traffic light [60], or detecting the road lines during driving [61],
- **Automatic video classification**: i.e., detecting videos and categorising based on content [62],
- **Object Detection:** Such as detecting objects in an image [20].

In addition, CNNs are not restricted to visual perception:

> They are also successful at many other tasks, such as voice recognition and natural language processing.

However, as our topic is **Image Processing**, we will focus on its visual applications. In this chapter we will explore where CNNs came from, what their building blocks look like, and how to implement them using `tf.keras`. Then we will discuss some of the best CNN architectures, as well as other visual tasks, including object detection[2] and semantic segmentation..

[2] classifying multiple objects in an image and placing bounding boxes around them

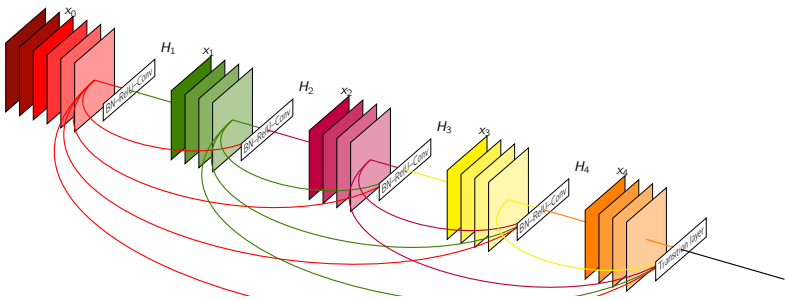


Figure 7.1: A standard CNN architecture. Don't worry if it looks too confusing at the moment as by the end of this chapter we will have the knowledge to understand and build your very own CNN.

7.2 Visual Cortex Architecture

David H. Hubel and *Torsten Wiesel* performed a series of experiments on cats in 1958 and 1959 (and a few years later on monkeys), giving crucial insights into the structure of the visual cortex[3] [63]. What is important to us is, they showed that many neurons in the visual cortex have a small local receptive field, meaning they react only to visual stimuli located in a limited region of the visual field. A diagram showcasing this phenomenon can be seen in **Fig.** 7.2, in which the local receptive fields of five neurons are represented by dashed circles. The receptive fields of different neurons may overlap, and together they tile the whole visual field.

[3] the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work

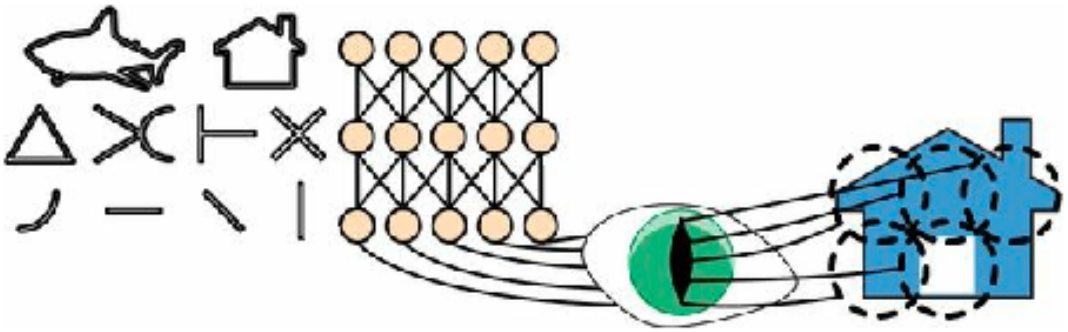


Figure 7.2: Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields

In addition to the previous revelation of how neurons work, they showed some neurons react **ONLY** to images of horizontal lines, while others react **ONLY** to lines with different orientations[4]. They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons. (in **Fig.** 7.2, observe that each neuron is connected only to nearby neurons from the previous layer).

[4] two neurons may have the same receptive field but react to different line orientations

This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field. These studies of the visual cortex inspired the neocognitron [64] , introduced in 1980, which gradually evolved into what we now call CNN.

An important milestone was a 1998 paper by *Yann LeCun et.al.* which introduced the famous LeNet-5 architecture, which became widely used by banks to recognize handwritten digits on checks [65]. This architecture has some building blocks were are familiar with:

- Fully connected layers,
- Sigmoid activation functions[5],

but it also introduces two new building blocks:

[5] A function allowing non-linear properties for the neural network.

- convolutional layers
- pooling layers.

Which will, of course, will be our focus of attention this chapter.

Information: Limits of DNN

Why not simply use a DNN with fully connected layers for image recognition tasks? Why do we need a new method ?

Unfortunately, although this works fine for small images (e.g., MNIST), it breaks down for larger images due to the **huge number of parameters** it requires.
For example, a 100-by-100 pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer.

CNNs solve this problem using partially connected layers and weight sharing.

7.3 Convolutional Layers

The most important building block of a CNN is the convolutional layer:

> Neurons in the first convolutional layer are **NOT** connected to every single pixel in the input image[6].

[6] This approach is going against the previously discussed ANNs and DNNs

The neurons are only connected to pixels in their receptive fields which its graphical representation can be seen in **Fig.** 7.3.

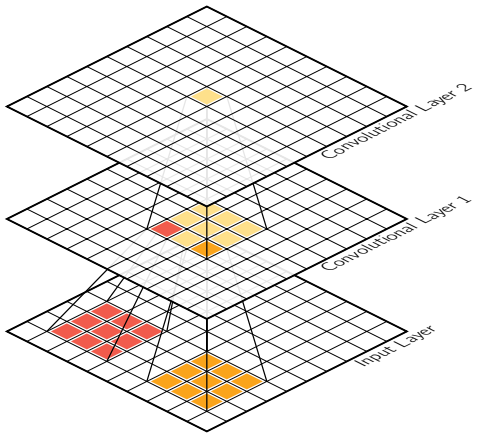


Figure 7.3: CNN layers with rectangular local receptive fields.

> In a CNN, each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs as our images are also 2D.

In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field which can be observed in **Fig. ??**.

> For a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, which is called *zero padding*.

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, shown in **Fig. ??**.

This significantly decreases the model's computational complexity. The horizontal or vertical step

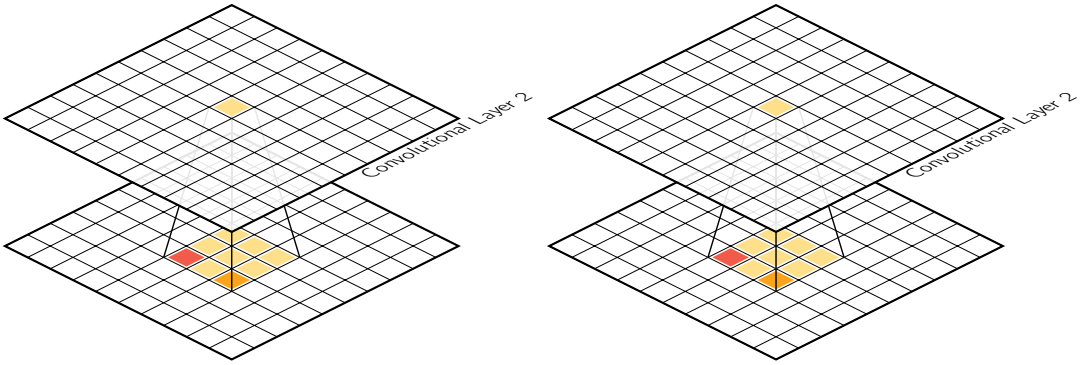


Figure 7.4

size from one receptive field to the next is called the **stride**. In **Fig. ??**, a 5-by-7 input layer (plus zero padding) is connected to a 3-by-4 layer, using 3-by-3 receptive fields and a stride of 2[7]. A neuron located in row *i*, column *j* in the upper layer is connected to the outputs of the neurons in the previous layer located in rows iŒsh toiŒsh +fh -1,columnsjŒsw to j Œ sw + fw - 1, where sh and sw are the vertical and horizontal strides.

[7] in this example the stride is the same in both directions, but it does not have to be so

7.3.1 Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, **Fig.** 7.5 shows two (2) possible sets of weights, called filters[8].

[8] The terms convolution kernels, or kernels are also used

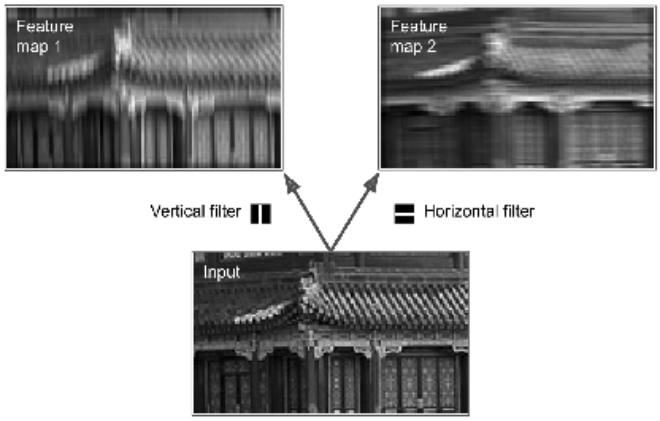


Figure 7.5: An example image showcasing the effect of applying filters to get feature maps.

The first one is represented as a black square with a vertical white line in the middle.

> It's a 7-by-7 matrix full of 0s except for the central column, which is full of 1s.

Neurons using these weights will ignore everything in their receptive field except for the central vertical line[9]. The second filter is a black square with a horizontal white line in the middle. Neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

[9] since all inputs will be multiplied by 0, except for the ones in the central vertical line

If all neurons in a layer use the same vertical line filter (and the same bias term), and we feed the network the input image shown in **Fig.** 7.5 (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what we get if all neurons use the same horizontal line filter. Notice the horizontal white lines get enhanced while the rest is blurred out. Therefore, a layer full of neurons using the same filter outputs a feature map, which highlights the areas in an image that activate the filter the most.

> We won't have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

7.3.2 Stacking Multiple Feature Maps

Up to now, for simplicity, We represented the output of each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters[10] and outputs one feature map per filter, so it is more accurately represented in 3D, which can be seen in **Fig.** 7.6.

[10] The number of filters is left up to the programmer

It has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same kernel and bias term). Neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the feature maps of the previous layer. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

> The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model. Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a fully connected neural network has learned to recognize a pattern in one location, it can only recognize it in that particular location.

Input images are also composed of multiple sublayers: one per color channel. As we already now, there are typically three: red, green, and blue (RGB). Grayscale images have just one channel, but some images may have many more-for example, satellite images that capture extra light frequencies

Specifically, a neuron located in row i, column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l-1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$,across all feature maps(in layer $l-1$).

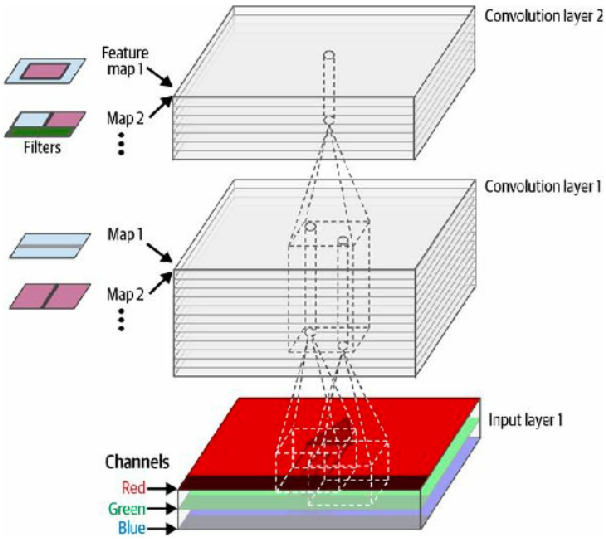


Figure 7.6

Within a layer, all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

This definition can be summarised in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer.

Let's try to understand the variables:

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- s_h, s_w are the vertical and horizontal strides, f_h, f_w are the height and width of the receptive field, and f_n' is the number of feature maps in the previous layer ($l-1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l-1$, row i', column j', feature map k', or channel k' if the previous layer is the input layer.
- b_k is the bias term for feature map k (in layer l). Think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

7.3.3 Implementing Convolutional Layers with Keras

As with every ML application, we load and preprocess a couple of sample images, using `sklearn`'s `load_sample_image()` function and Keras's CenterCrop and Rescaling layers:

C.R. 1 python
```
from sklearn.datasets import load_sample_images
import tensorflow as tf

images = load_sample_images()["images"]
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)
```

Let's look at the shape of the images tensor[11]:

C.R. 2 python
```
print(images.shape)
```

C.R. 3 text
```
(2, 70, 120, 3)
```

[11] A tensor is a generalization of vectors and matrices to potentially higher dimensions.

It's a 4D tensor. Let's see why this is a 4D matrix. There are two (2) sample images, which explains the first dimension. Then each image is 70-by-120, as that's the size we specified when creating the CenterCrop layer[12]. This explains the second and third dimensions. And lastly, each pixel holds one value per colour channel, and there are three (3) of them:

Red, Green, and Blue.

[12] the original images were 427-by-640

Now let's create a 2D convolutional layer and feed it these images to see what comes out. For this, Keras provides a Convolution2D layer, alias `Conv2D`. Under the hood, this layer relies on TensorFlow's `tf.nn.conv2d()` operation.

Let's create a convolutional layer with 32 filters, each of size 7-by-7 (using `kernel_size=7`, which is equivalent to using `kernel_size=(7 , 7)`), and apply this layer to our small batch of two (2) images:

C.R. 4 python
```
tf.random.set_seed(42)  # extra code  ensures reproducibility
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7)
fmaps = conv_layer(images)
```

Now let's look at the output's shape:

C.R. 5 python
```
print(fmaps.shape)
```

C.R. 6 text
```
(2, 64, 114, 32)
```

The output shape is similar to the input shape, with two (2) main differences.

1. There are 32 channels instead of 3. This is because we set `filters=32`, so we get 32 output feature maps: instead of the intensity of red, green, and blue at each location, we now have the intensity of each feature at each location.

2. The height and width have both shrunk by 6 pixels. This is due to the fact that the Conv2D layer does **NOT** use any zero-padding by default, which means that we lose a few pixels on the sides of the output feature maps, depending on the size of the filters. In this case, since the kernel size is 7, we lose 6 pixels horizontally and 6 pixels vertically (i.e., 3 pixels on each side).

If instead we set `padding="same"`, then the inputs are padded with enough zeros on all sides to ensure that the output feature maps end up with the same size as the inputs (hence the name of this option):

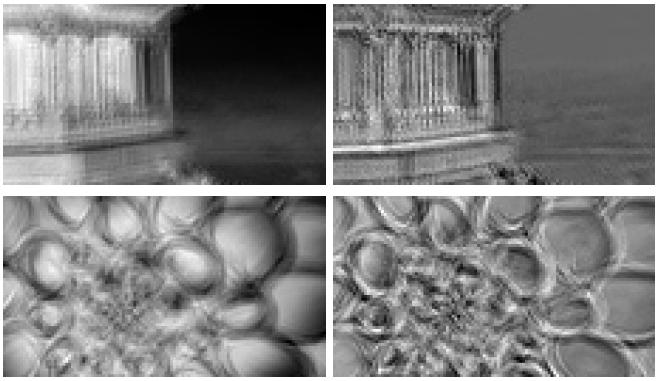


Figure 7.7

These two padding options are illustrated in **Fig.** 7.8. For simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension as well.

If the stride is greater than 1 (in any direction), then the output size will not be equal to the input size, even if `padding="same"`. For example, if we set `strides=2` (or equivalently `strides=(2, 2)`), then the output feature maps will be 35-by-60: halved both vertically and horizontally.

Fig. 7.8 shows what happens when **strides=2**, with both padding options.

If we are curious, this is how the output size is computed:

- With `padding="valid"`, if the width of the input is ih, then the output width is equal to (ih - fh + sh) / sh, rounded down. Recall that fh is the kernel width, and sh is the horizontal stride. Any remainder in the division corresponds to ignored columns on the right side of the input image. The same logic can be used to compute the output height, and any ignored rows at the bottom of the image.

- With `padding="same"`, the output width is equal to ih / sh, rounded up. To make this possible, the appropriate number of zero columns are padded to the left and right of the input image (an equal number if possible, or just one more on the right side). Assuming the output

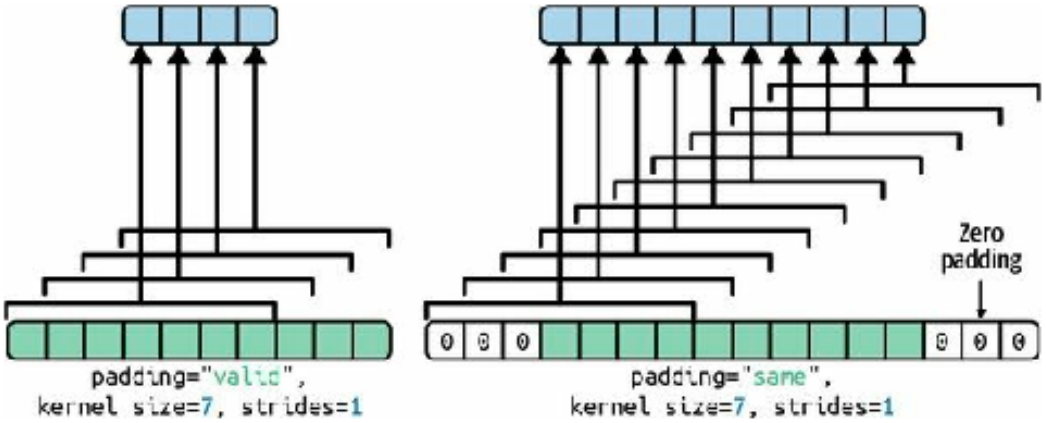


Figure 14-7. The two padding options, when strides=1

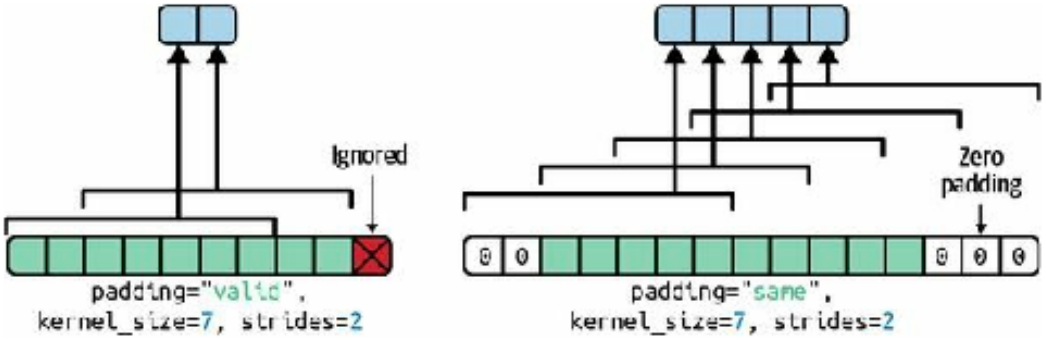


Figure 7.8: Showcasing different padding options.

width is ow, then the number of padded zero columns is (ow - 1) Œ sh + fh - ih. Again, the same logic can be used to compute the output height and the number of padded rows.

Now let's look at the layer's weights (which were noted wu, v, k', k and bk in Equation 14-1). Just like a Dense layer, a Conv2D layer holds all the layer's weights, including the kernels and biases. The kernels are initialized randomly, while the biases are initialized to zero. These weights are accessible as TF variables via the weights attribute, or as NumPy arrays via the `get_weights()` method:

The kernels array is 4D, and its shape is [`kernel_height`, `kernel_width`, `input_channels`, `output_channels`]. The biases array is 1D, with shape [`output_channels`]. The number of output channels is equal to the number of output feature maps, which is also equal to the number of filters. Most importantly, note that the height and width of the input images do not appear in the kernel's shape: this is because all the neurons in the output feature maps share the same weights, as explained earlier. This means that we can feed images of any size to this layer, as long as they are at least as large as the kernels, and if they have the right number of channels (three in this case).

Lastly, we will generally want to specify an activation function (such as ReLU) when creating a Conv2D layer, and also specify the corresponding kernel initializer. This is for the same reason as for Dense layers: a convolutional layer performs a linear operation, so if we stacked multiple convolutional layers without any activation functions they would all be equivalent to a single convolutional layer,

and they wouldn't be able to learn anything really complex.

As we can see, convolutional layers have quite a few hyperparameters: filters, `kernel_size`, padding, strides, activation, `kernel_initializer`, etc. As always, we can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later in this chapter, to give you some idea of which hyperparameter values work best in practice.

7.3.4 Memory Requirements

Another challenge with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of back-propagation requires all the intermediate values computed during the forward pass.

As an example, consider a convolutional layer with 200 5-by-5 filters, with stride 1 and `"same"` padding. If the input is a 150-by-100 RGB image, then the number of parameters is (5-by-5-by-3+1) Œ 200 = 15,200[13], which is fairly small compared to a fully connected layer. However, each of the 200 feature maps contains 150 Œ 100 neurons, and each of these neurons needs to compute a weighted sum of its 5-by-5-by-3 = 75 inputs: that's a total of 225 million float multiplications.

[13] the + 1 corresponds to the bias terms

Not as bad as a fully connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (12 MB) of RAM. 8 And that's just for one instance-if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM.

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so we only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

Now let's look at the second common building block of CNNs: the pooling layer.

7.4 Pooling Layer

The goal of a pooling layer is to subsample (i.e., shrink) the input image to reduce the computational load, the memory usage, and the number of parameters[14].

[14] This limits the risk of overfitting

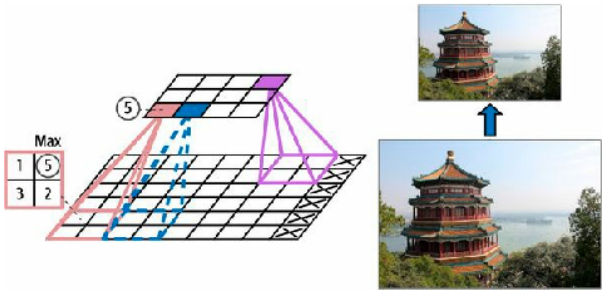


Figure 7.9

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. We must define its size, the stride, and the padding type, just like before.

However, a pooling neuron has **NO** weights. All it does is aggregate the inputs using an aggregation function such as the max or mean.

Fig. 7.9 shows a **max pooling layer**, which is the most common type of pooling layer. In this example, we use a 2-by-2 pooling kernel, with a stride of 2 and no padding. Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field in **Fig.** 7.9, the input values are 1, 5, 3, 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

> A pooling layer typically works on every input channel independently, so the output depth (i.e., the number of channels) is the same as the input depth.

Other than reducing computations, memory usage, and the number of parameters, a max pooling layer also introduces some level of invariance[15] to small translations, as shown in **Fig.** 7.10.

[15] In the context of computer vision, it means that we can recognize an object as an object, even when its appearance varies in some way.

Here we assume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2-by-2 kernel and stride 2. Images B and C are the same as image A, but shifted by one and two pixels to the right. As we can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. For image C, the output is different: it is shifted one pixel to the right (but there is still 50% invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale.

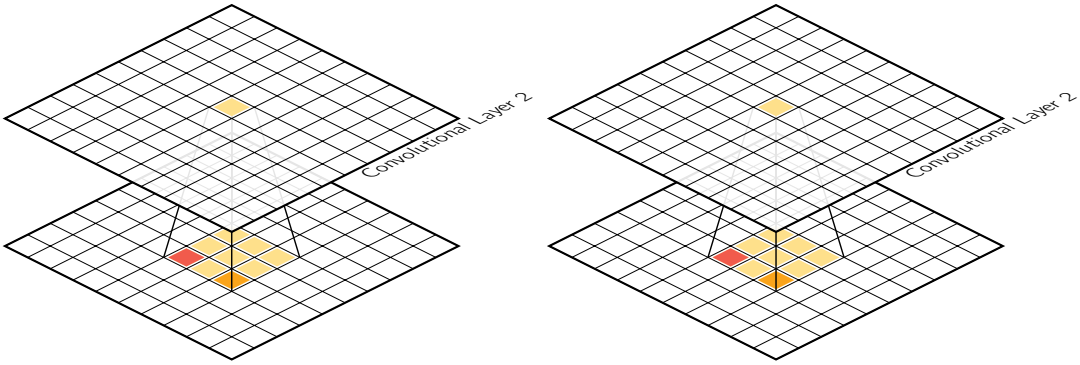


Figure 7.10: As can be seen from the image above, max-pooling allow a certain level of invariance when the object moves in small increments. This is an important property as it is favourable when small changes in movement don't affect the overall recognition of the image.

> Moreover, max pooling offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

However, max pooling has some downsides too. It's obviously very destructive: even with a tiny 2 Œ 2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values. And in some applications, invariance is not desirable. Take semantic segmentation[16].

[16] The task of classifying each pixel in an image according to the object that pixel belongs to

For this case, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is equivariance, not invariance: a small change to the inputs should lead to a coresponding small change in the output.

7.5 Implementing Pooling Layers with Keras

The following code creates a MaxPooling2D layer, alias **MaxPool2D**, using a 2-by-2 kernel. The strides default to the kernel size, so this layer uses a stride of 2 (horizontally and vertically). By default, it uses **"valid"** padding (i.e., no padding at all):

C.R. 7
```python
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

To create an average pooling layer, just use AveragePooling2D, alias **AvgPool2D**, instead of MaxPool2D. As we might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. Average pooling layers used to be very popular, but people mostly use max pooling layers now, as they generally perform better. This may seem surprising, since computing the mean generally loses less information than computing the max. But on the other hand, max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with. Moreover, max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Note that max pooling and average pooling can be performed along the depth dimension instead of the spatial dimensions, although it's not as common. This can allow the CNN to learn to be invariant to various features. For example, it could learn multiple filters, each detecting a different rotation of the same pattern (such as handwritten digits seen in **Fig.** 7.11), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation. The CNN could similarly learn to be invariant to anything: thickness, brightness, skew, color, and so on.

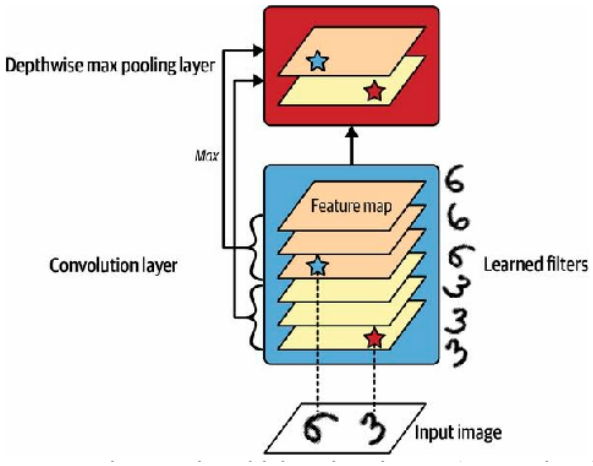


Figure 7.11: Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case).

Keras does not include a depthwise max pooling layer, but it's not too difficult to implement a custom layer for that:

C.R. 8 python

```python
class DepthPool(tf.keras.layers.Layer):
    def __init__(self, pool_size=2, **kwargs):
        super().__init__(**kwargs)
        self.pool_size = pool_size

    def call(self, inputs):
        shape = tf.shape(inputs)  # shape[-1] is the number of channels
        groups = shape[-1] // self.pool_size  # number of channel groups
        new_shape = tf.concat([shape[:-1], [groups, self.pool_size]], axis=0)
        return tf.reduce_max(tf.reshape(inputs, new_shape), axis=-1)
```

This layer reshapes its inputs to split the channels into groups of the desired size (`pool_size`), then it uses `tf.reduce_max()` to compute the max of each group. This implementation assumes that the stride is equal to the pool size, which is generally what we want. Alternatively, we could use TensorFlow's `tf.nn.max_pool()` operation, and wrap in a Lambda layer to use it inside a Keras model, but sadly this op does not implement depthwise pooling for the GPU, only for the CPU.

One last type of pooling layer that we will often see in modern architectures is the global average pooling layer. It works very differently: all it does is compute the mean of each entire feature map (it's like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs). This means that it just outputs a single number per feature map and per instance. Although this is of course extremely destructive (most of the information in the feature map is lost), it can be useful just before the output layer, as we will see later in this chapter. To create such a layer, simply use the GlobalAveragePooling2D class, alias GlobalAvgPool2D:

7.6 CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the convolutional layers (see **Fig. ??**). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

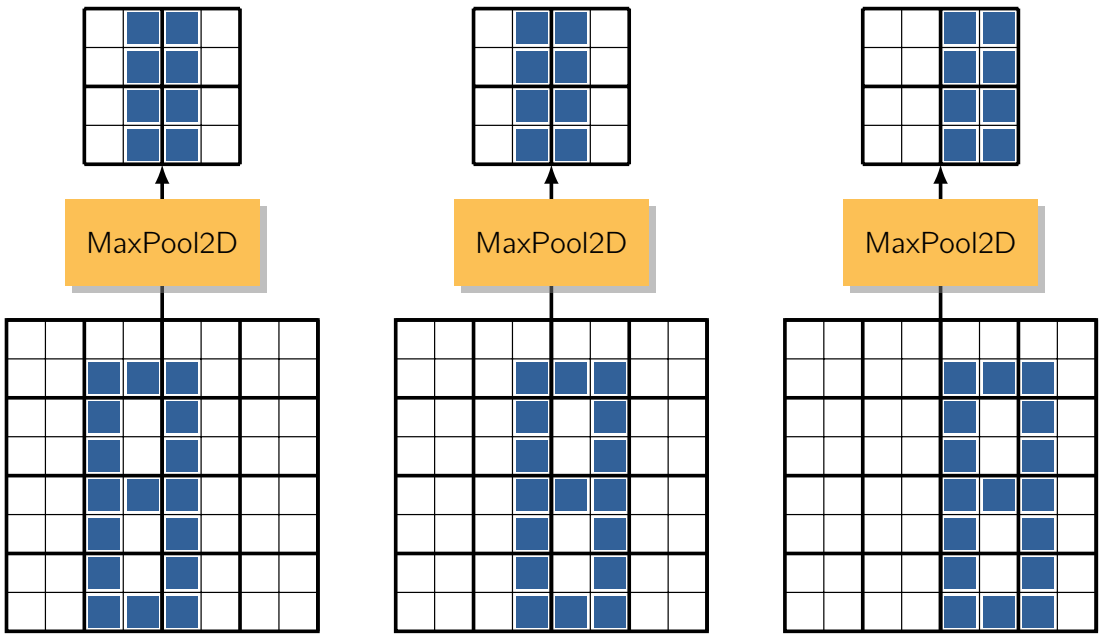


Figure 7.12: An example structure of a CNN network

A common mistake is to use convolution kernels that are too large. For example, instead of using a convolutional layer with a 5 Œ 5 kernel, stack two layers with 3 Œ 3 kernels: it will use fewer parameters and require fewer computations, and it will usually perform better. One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5 Œ 5), usually with a stride of 2 or more. This will reduce the spatial dimension of the image without losing too much information, and since the input image only has three channels in general, it will not be too costly.

Here is how we can implement a basic CNN to tackle the Fashion MNIST dataset.

Let's first download and allocated the traning/testing.

C.R. 9

```python
import numpy as np

mnist = tf.keras.datasets.fashion_mnist.load_data()
```

C.R. 10 python
```
(X_train_full, y_train_full), (X_test, y_test) = mnist
X_train_full = np.expand_dims(X_train_full, axis=-1).astype(np.float32) / 255
X_test = np.expand_dims(X_test.astype(np.float32), axis=-1) / 255
X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]
```

C.R. 11 python
```
from functools import partial

tf.random.set_seed(42)  # extra code  ensures reproducibility
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding="same",
                        activation="relu", kernel_initializer="he_normal")
model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=10, activation="softmax")
])
```

Let's go through this code:

1. We use the `functools.partial()` function to define DefaultConv2D, which acts just like Conv2D but with different default arguments: a small kernel size of 3, `"same"` padding, the ReLU activation function, and its corresponding *He initializer*.

2. We create the Sequential model. Its first layer is a DefaultConv2D with 64 fairly large filters (7-by-7). It uses the default stride of 1 because the input images are not very large. It also sets `input_shape=[28, 28, 1]`, as the images are 28-by-28 pixels, with a single color channel (i.e., grayscale).

3. When we load the Fashion MNIST dataset, we need to make sure each image has this shape. Therefore we may require to use `np.reshape()` or `np.expanddims()` to add the channels dimension or we could use a Reshape layer as the first layer in the model.

4. We then add a max pooling layer that uses the default pool size of 2, so it divides each spatial dimension by a factor of 2.

5. We repeat the same structure twice: two convolutional layers followed by a max pooling

layer. For larger images, we could repeat this structure several more times. The number of repetitions is a hyperparameter we can tune

> Observe the number of filters doubles as we climb up the CNN toward the output layer (it is initially 64, then 128, then 256): it makes sense for it to grow, since the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many different ways to combine them into higher-level features. It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2, we can afford to double the number of feature maps in the next layer without fear of exploding the number of parameters, memory usage, or computational load.

6. Next is the fully connected network, composed of two hidden dense layers and a dense output layer. Since it's a classification task with 10 classes, the output layer has 10 units, and it uses the softmax activation function. Note that we must flatten the inputs just before the first dense layer, since it expects a 1D array of features for each instance. We also add two dropout layers, with a dropout rate of 50% each, to reduce overfitting.

If we compile this model using the `"sparse_categorical_crossentropy"` loss and we fit the model to the Fashion MNIST training set, it should reach over 92% accuracy on the test set.

C.R. 12

```python
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
score = model.evaluate(X_test, y_test)
X_new = X_test[:10]  # pretend we have new images
y_pred = model.predict(X_new)
```

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC ImageNet challenge. In this competition, the top-five error rate for image classification -that is, the number of test images for which the system's top five predictions did not include the correct answer-fell from over 26% to less than 2.3% in just six years. The images are fairly large (e.g., 256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds).

Looking at the evolution of the winning entries is a good way to understand how CNNs work, and how research in deep learning progresses. We will first look at:

- LeNet-5 architecture (1998)
- AlexNet (2012),
- GoogLeNet (2014),

- ResNet (2015),
- SENet (2017).

In addition, we will also briefly look at more architectures, including Xception, ResNeXt, DenseNet, MobileNet, CSPNet, and EfficientNet.

7.6.1 LeNet-5

The LeNet-5 architecture is perhaps the most widely known CNN architecture. As mentioned, it was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST).

Layer	Type	Maps	Size	Kernel Size
Out	Fully Connected	-	10	-
F6	Fully connected	-	84	-
C5	Convolution	120	1-by-1	5-by-5
S4	Average Pooling	16	5-by-5	2-by-2
C3	Convolution	16	10-by-10	5-by-5
S2	Average Pooling	6	14-by-14	2-by-2
C1	Convolution	6	28-by-28	5-by-5
In	Input	1	32-by-32	-

Table 7.1: LeNet-5 Architecture.

As we can see, this looks pretty similar to our Fashion MNIST model: a stack of convolutional layers and pooling layers, followed by a dense network. Perhaps the main difference with more modern classification CNNs is the activation functions: today, we would use ReLU instead of tanh and softmax instead of RBF.

7.6.2 AlexNet

The AlexNet CNN architecture won the 2012 ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best competitor achieved only 26%! AlexaNet was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer. Table X presents this architecture.

To reduce overfitting, the authors used two regularization techniques. First, they applied dropout with a 50% dropout rate during training to the outputs of layers F9 and F10. Second, they

performed data augmentation by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

Information: Data Augmentation

It is the process of **artificially increasing** the size of the training set by generating many realistic **variants** of each training instance. This process aims to reduce over-fitting, making this a regularisation technique.

The generated instances should be as realistic as possible. This means, in an ideal scenario, a human should not be able to tell whether it was augmented or not. In adition, it has to be something **learnable**; Adding white noise will not help as it is a random process and there is no pattern in the data for the algorithm to learn.

For example, we can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set.

Figure 7.13: Data augmentation is the process of artificially generating new data from existing data. Here can see the process where the original image is transformed (shear, rotation) and fed to the ML to train on this new data. This allows the reuse of the image without requiring to gather new data [66].

To use this feature in code, use Keras's data augmentation layers, (i.e., `RandomCrop`, `RandomRotation`, etc.). These layers force the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. To produce a model that's more tolerant of different lighting conditions, we can similarly generate many images with **various contrasts**. In general, we can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, we can greatly increase your training set size. Example of this operation can be seen in **Fig.** 7.13.

Data augmentation is also useful when we have an unbalanced dataset: we can use it to generate more samples of the less frequent classes. This is called the synthetic minority oversampling technique (SMOTE).

AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called *local response normalization* (LRN):

> The most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps.

Such competitive activation has been observed in biological neurons [67]. This encourages different

feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization. The following equation gives use a view on how to apply this method:

$$b_i = a_i \left(k + \alpha \sum_{j=j_{low}}^{j_{high}} a_j^2 \right)^{-\beta} \quad \text{with} \quad \begin{cases} j_{high} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{low} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

Let's discuss what each parameter means:

- b_i is the neuron's normalised output located in feature map i, at some row u and column v[17].
- a_i is the activation of that neuron after the ReLU step, but before normalisation.
- k, α, β and r are hyperparameters. k is called the bias, and r is called the depth radius.
- f_n is the number of feature maps.

[17] In this equation we consider only neurons located at this row and column, so u and v are not shown

To give an example, if $r = 2$ and a neuron has a **strong activation**, it will inhibit the activation of the neurons located in the feature maps immediately above and below its own.

In AlexNet, the hyperparameters are set as:

$$r = 5,\ \alpha = 0.0001,\ \beta = 0.75 \quad \text{and} \quad k = 2.$$

We can implement this step by using the `tf.nn.local_response_normalization()` function.

> A variant of AlexNet called ZF Net was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge [68]. It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

7.6.3 GoogLeNet

The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research, and won the ILSVRC 2014 challenge by pushing the top-five error rate below 7% [69].

This performance boost came in from the network being deeper than previous CNNs. This was made possible by subnetworks called inception modules, which allow GoogLeNet to use parameters much more efficiently than previous architectures:

> GoogLeNet actually has 10 times fewer parameters than AlexNet[18].

[18] roughly 6 million instead of 60 million

Figure 14-14 shows the architecture of an inception module. The notation "$3 \times 3 + 1\,(S)$" means the layer uses a 3-by-3 kernel, stride 1, and `same` padding. The input signal is first fed to four (4) different layers in parallel. All convolutional layers use the ReLU activation function.

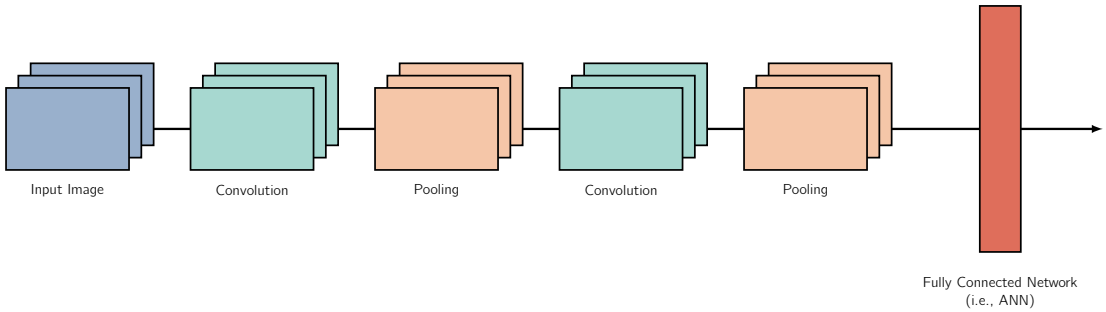


Figure 7.14: The GoogLeNet architecture. The nodes coloured in (■) are called inception nodes.

> Top convolutional layers use different kernel sizes (1-by-1, 3-by-3, and 5-by-5), allowing them to capture patterns at different scales.

Also note that every single layer uses a stride of 1 and "same" padding[19]. This is done so outputs all have the same height and width as their inputs. This allows concatenating all outputs along the depth dimension in the final depth concatenation layer (i.e., to stack the feature maps from all four top convolutional layers).

[19] even the max pooling layer

> It can be implemented using Keras's `Concatenate` layer, using the default `axis=-1`.

You may wonder why inception modules have convolutional layers with 1-by-1 kernels.

> Surely these layers cannot capture any features because they look at only one pixel at a time, right?

In fact, these layers serve three (3) purposes:

1. Although they cannot capture spatial patterns, they can capture patterns along the depth dimension (i.e., across channels).
2. They are configured to output fewer feature maps compared to their inputs, so they serve as **bottleneck layers**, meaning they reduce dimensionality. This cuts the computational cost and the number of parameters, speeding up training and improving generalization.
3. Each pair of convolutional layers ([1-by-1, 3-by-3] and [1-by-1, 5-by-5]) acts like a single powerful convolutional layer, capable of capturing more complex patterns. A convolutional layer is equivalent to sweeping a dense layer across the image (at each location, it only looks at a small receptive field), and these pairs of convolutional layers are equivalent to sweeping two-layer neural networks across the image.

In short, we can think of the whole inception module as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales.

Now let's look at the architecture of the GoogLeNet CNN shown in **Fig.** 7.14. The number of

feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. The architecture is so deep that it has to be represented in three (3) columns, but GoogLeNet is actually one tall stack, including nine (9) inception modules (nodes coloured in ■). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module.

All the convolutional layers use the ReLU activation function.

Let's go through this network together:

- The first two (2) layers divide the image's height and width by 4 (so its area is divided by 16). This reduces the computational load. The first layer uses a large kernel size, 7-by-7, so that much of the information is preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features .

Information: Local Normalisation Layer

This layer's job is to create a sort of **lateral inhibition**. This refers to the capacity of an excited neuron to subdue its neighbors. We basically want a significant peak so that we have a form of local maxima. This tends to create a contrast in that area, hence increasing the sensory perception.

- Two convolutional layers follow, where the first acts like a bottleneck layer. Think of this pair as a single smarter convolutional layer.
- A local response normalisation layer ensures the previous layers capture a wide variety of patterns.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the CNN's backbone: a tall stack of nine (9) inception modules, interleaved with a couple of max pooling layers to reduce dimensionality and speed up the net.
- Next, the global average pooling layer outputs the mean of each feature map: this drops any remaining spatial information, which is fine because there is not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224 Œ 224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7 Œ 7.

This classification task, not localization, so it doesn't matter where the object is.

Thanks to the dimensionality reduction brought by the global average pool layer, there is no need to have several fully connected layers at the top of the CNN[20], and this considerably reduces the number of parameters in the network and limits the risk of overfitting.

[20] This is unlike AlexNet.

- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

The original GoogLeNet architecture included two auxiliary classifiers plugged on top of the third and sixth inception modules. They were both composed:

- One average pooling layer
- one convolutional layer
- two fully connected layers
- a softmax activation layer

During training, their loss (scaled down by 70%) was added to the overall loss.

> The goal adding these auxiliary classifiers was to fight the vanishing gradients problem and regularize the network, but it was later shown that their effect was relatively minor.

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 [70] and Inception-v4 [71], using slightly different inception modules to reach even better performance.

7.6.4 VGGNet

The runner-up in the ILSVRC 2014 challenge was VGGNet [72], Karen Simonyan and Andrew Zisserman, from the Visual Geometry Group (VGG) research lab at Oxford University, developed a very simple and classical architecture; it had 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on, reaching a total of 16 or 19 convolutional layers, depending on the VGG variant. To add to this stack a final dense network with 2 hidden layers and the output layer. It used small 3-by-3 filters, but it had many of them.

7.6.5 ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a Residual Network (ResNet) that delivered an astounding top-five error rate under 3.6%. The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers) [73].

> Computer vision models are getting deeper and deeper, with fewer and fewer parameters.

The key idea for training such a deep network is to use skip connections[21].

[21] This is also called shortcut connections.

The signal feeding into a layer is also added to the output of a layer located higher up the stack.

Let's see why this is useful. When training a neural network, the goal is simple:

To make it model a target function $h(x)$.

If we add the input x to the output of the network (i.e., we add a skip connection), then the network will be forced to model:

$$f(x) = h(x) - x \qquad \text{rather than} \qquad h(x)$$

This approach is called residual learning [73].

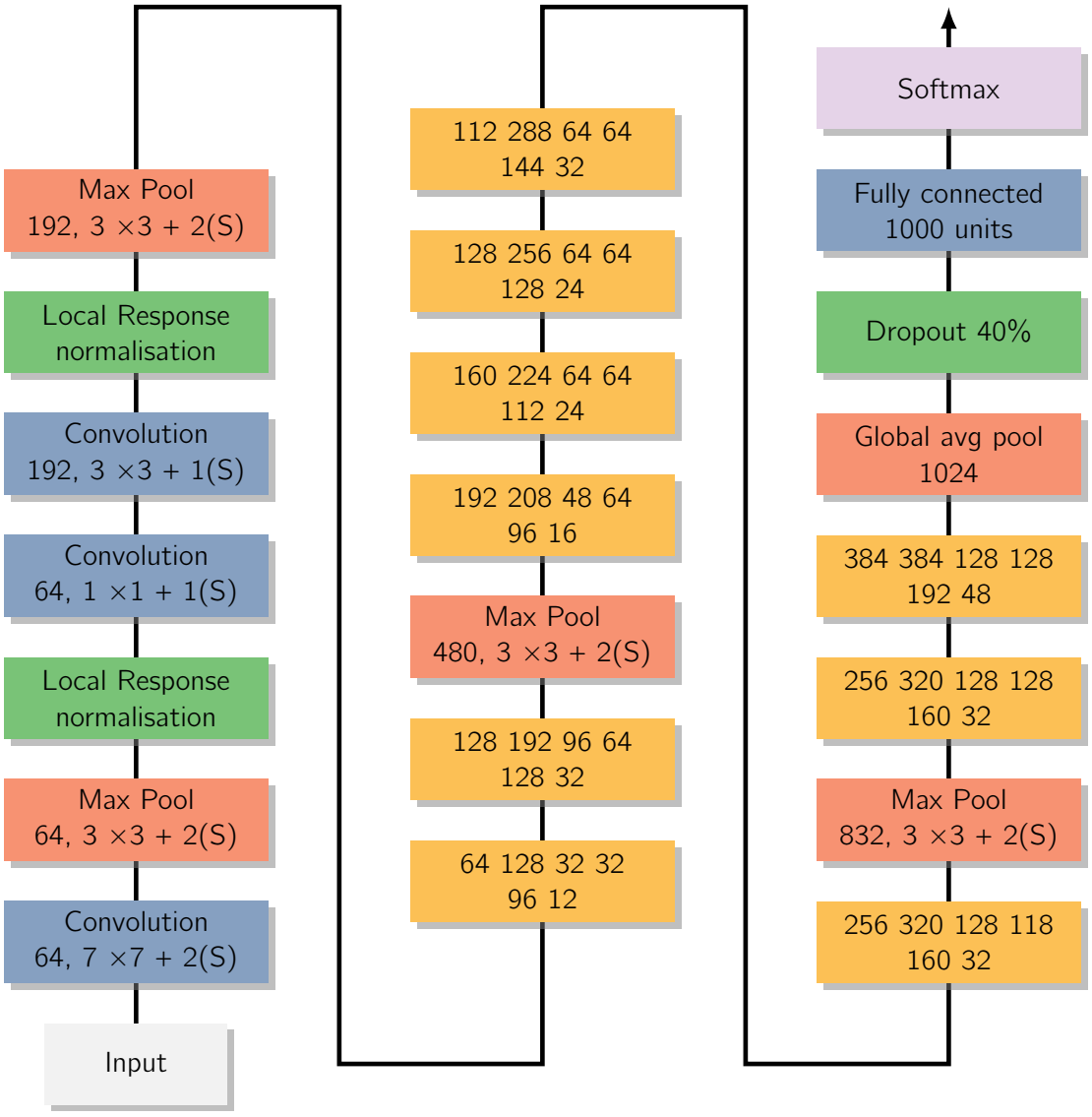


Figure 7.15

When we initialise a regular neural network, its weights are close to zero[22], so the network just

[22] They are not exactly zero but randomly assigned and are close to zero

outputs values close to zero. If we add a skip connection, the resulting network just outputs a copy of its inputs. In other words, it initially models the identity function.

> If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

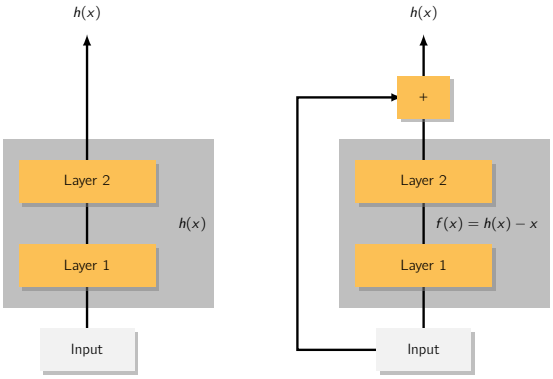


Figure 7.16

In addition to the previously mentioned positive aspects, if we add many skip connections, the network can start making progress even if several layers have not started learning yet [74], which we can see the diagram in **Fig.** 7.15.

> Skip connections allows the signal to easily make its way across the whole network.

The deep residual network can be seen as a stack of residual units (RUs), where each residual unit is a small neural network with a skip connection. Now let's look at ResNet's architecture (see Figure 14-18).

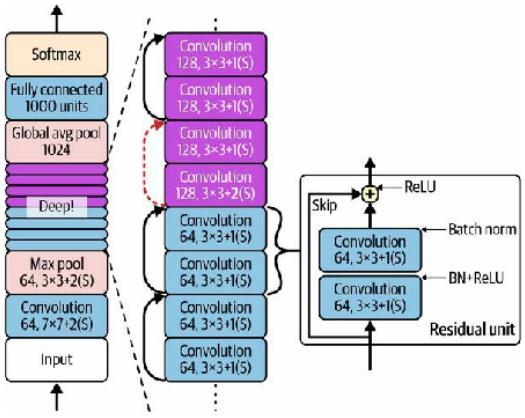


Figure 7.17

The idea of ResNet is simple to describe. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of residual units. Each residual unit is

composed of two (2) convolutional layers[23], with batch normalization (BN) and ReLU activation, using 3-by-3 kernels and preserving spatial dimensions (stride of 1, `"same"` padding).

[23] There are no pooling layers

> The number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2)

When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in Figure 14-18). To solve this problem, the inputs are passed through a 1-by-1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 14-19).

There are different variations of this aforementioned architecture, each having different numbers of layers. ResNet-34, as the name implies, is a ResNet with 34 layers (only counting the convolutional layers and the fully connected layer) containing 3 RUs that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps.

ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two (2) 3-by-3 convolutional layers with 256 feature maps, they use three (3) convolutional layers:

1. a 1-by-1 convolutional layer with just 64 feature maps (4x less), which acts as a bottleneck layer (as discussed already)
2. a 3-by-3 layer with 64 feature maps
3. another 1-by-1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth

ResNet-152 contains 3 such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.

7.7 Implementing a ResNet-34 CNN using Keras

Most CNN architectures described so far can be implemented easily using Keras[24]. To illustrate the process, let's implement a ResNet-34 from scratch with Keras.

[24] although generally we would load a pre-trained network instead, as we will see later.

First, we'll create a `ResidualUnit` layer:

C.R. 13

```python
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, strides=1,
                        padding="same", kernel_initializer="he_normal",
                        use_bias=False)

class ResidualUnit(tf.keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            tf.keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            tf.keras.layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                tf.keras.layers.BatchNormalization()
            ]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)

```

As we can see, this code matches Figure 14-19 pretty closely. In the constructor, we create all the layers we will need:

> the main layers are the ones on the right side of the diagram, and the skip layers are the ones on the left[25].

[25] only needed if the stride is greater than 1.

Then in the `call()` method, we make the inputs go through the main layers and the skip layers (*if any*), and we add both outputs and apply the activation function.

Now we can build a ResNet-34 using a Sequential model, as it's really just a long sequence of layers. We can treat each residual unit as a single layer now that we have the `ResidualUnit` class.

The code closely matches:

C.R. 14
```python
model = tf.keras.Sequential([
    DefaultConv2D(64, kernel_size=7, strides=2, input_shape=[224, 224, 3]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"),
])
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters

model.add(tf.keras.layers.GlobalAvgPool2D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

The only tricky part in this code is the loop that adds the `ResidualUnit` layers to the model. As explained earlier:

- the first 3 RUs have 64 filters,
- the next 4 RUs have 128 filters.

and so on. At each iteration, we must set the stride to 1 when the number of filters is the same as in the previous RU, or else we set it to 2; then we add the ResidualUnit, and finally we update `prev_filters`.

With just 40 lines of code, we can build the model that won the ILSVRC 2015 challenge. This demonstrates both the elegance of the ResNet model and the expressiveness of the Keras API. Implementing the other CNN architectures is a bit longer, but not much harder.

However, Keras comes with several of these architectures built in, so why not use them instead?

7.8 Using Pre-Trained Models from Keras

In general, we don't have to implement standard models like GoogLeNet or ResNet manually, as pre-trained networks are readily available using the `tf.keras.applications` package.

For example, we can load the ResNet-50 model, pre-trained on ImageNet, with the following line of code:

C.R. 15 python
```
import tensorflow as tf

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

This was surprisingly simple. This will create a ResNet-50 model and download weights already trained on the ImageNet dataset. To use it, we first need to ensure the images have the correct size. A ResNet-50 model expects an image with the dimensions of 224-by-224-pixel[26], so let's use the Resizing layer, provided by Keras, to resize two (2) sample images (after cropping them to the target aspect ratio):

[26] other models may expect other sizes, such as 299-by-299.

C.R. 16 python
```
from sklearn.datasets import load_sample_images

K = tf.keras.backend
images = K.constant(load_sample_images()["images"])
images_resized = tf.keras.layers.Resizing(height=224, width=224,
                                          crop_to_aspect_ratio=True)(images)
```

The pre-trained models assumes the images are pre-processed in a specific way. In some cases the models can expect the inputs to be scaled from 0 to 1, or from -1 to 1, and so on. Each model provides a `preprocess_input()` function we can use to pre-process our images. These functions assume the original pixel values range from 0 to 255, which is the case here:

C.R. 17 python
```
inputs = tf.keras.applications.resnet50.preprocess_input(images_resized)
```

Now we can use the pre-trained model to make predictions:

C.R. 18 python
```
Y_proba = model.predict(inputs)
print(Y_proba.shape)
```

C.R. 19 text
```
(2, 1000)
```

As usual, the output `Y_proba` is a matrix with one row per image and one column per class[27]. To display the top K predictions, including the class name and the estimated probability of each predicted class, use the `decode_predictions()` function. For each image, it returns an array containing the top `K` predictions, where each prediction is represented as an array containing the class identifier, its name, and the corresponding confidence score:

[27] in this case, there are 1,000 classes

Figure 7.18: The images used in testing the image recognition.

C.R. 20

```python
top_K = tf.keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print(f"Image #{image_index}")
    for class_id, name, y_proba in top_K[image_index]:
        print(f"  {class_id} - {name:12s} {y_proba:.2%}")
```

The output looks like this:

C.R. 21

```text
Downloading data from
↪ https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
Image #0
  n03877845 - palace       54.69%
  n03781244 - monastery    24.71%
  n02825657 - bell_cote    18.55%
Image #1
  n04522168 - vase         32.67%
  n11939491 - daisy        17.82%
  n03530642 - honeycomb    12.04%
```

The correct classes are **palace** and **dahlia** (which you can see in **Fig.** 7.18), so the model is **correct for the first image but wrong for the second**.

> This is caused by dahlia not being part of the 1,000 ImageNet classes.

Keeping this in mind, vase is a reasonable guess[28], and daisy is also not a bad choice either, are dahlias and daisies both share similar features. As we can see, it is very easy to create a pretty good image classifier using a pre-trained model.

[28] The intricate design of the flower might be reinterpreted as a decoration painted on a vase

Many vision models are available in `tf.keras.applications`, from lightweight and fast models to large and accurate ones. But what if we want to use an image classifier for classes of images that

are not part of ImageNet? In that case, we may still benefit from the pre-trained models by using them to perform **transfer learning**.

7.9 Pre-Trained Models for Transfer Learning

If we want to build an image classifier but not have enough data to train it from scratch, it is often a good idea to reuse the lower layers of a pre-trained model [75]. For example, let's train a model to classify pictures of flowers, reusing a pre-trained Xception model.

First, we'll load the flowers dataset using TensorFlow Datasets:

C.R. 22 python
```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples
class_names = info.features["label"].names
n_classes = info.features["label"].num_classes
print(info)
```

> We can get information about the dataset by setting `with_info=True`.

Here, we get the dataset size and the names of the classes. Unfortunately, there is only a `"train"` dataset, no test set or validation set, so we need to split the training set. Let's call `tfds.load()` again, but this time taking the first 10% of the dataset for testing, the next 15% for validation, and the remaining 75% for training:

C.R. 23 python
```
test_set_raw, valid_set_raw, train_set_raw = tfds.load(
    "tf_flowers",
    split=["train[:10%]", "train[10%:25%]", "train[25%:]"],
    as_supervised=True)
```

All three (3) datasets contain individual images. First let's have a look at the data to get some idea on what we are working with.

We need to batch them, but first we need to ensure they all have the same size, otherwise batching will fail. We can use a `Resizing` layer for this. We must also call the `tf.keras.applications.xception.preprocess_input()` function to preprocess the images appropriately for the Xception model. Lastly, we'll also shuffle the training set and use prefetching:

C.R. 24 python
```
batch_size = 32
preprocess = tf.keras.Sequential([
    tf.keras.layers.Resizing(height=224, width=224, crop_to_aspect_ratio=True),
    tf.keras.layers.Lambda(tf.keras.applications.xception.preprocess_input)
])
train_set = train_set_raw.map(lambda X, y: (preprocess(X), y))
train_set = train_set.shuffle(1000, seed=42).batch(batch_size).prefetch(1)
valid_set = valid_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
test_set = test_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
```

Figure 7.19: Sample images present in the dataset dataset. As you can see the images are not all in the same shape which we need to work on.

Now each batch contains 32 images, all of them 224-by-224 pixels, with pixel values ranging from -1 to 1.

This is the ideal values for training such a network. As the dataset is not very large, a bit of data augmentation will certainly help. Let's create a data augmentation model that we will embed in our final model. During training, it will randomly flip the images horizontally, rotate them a little bit, and tweak the contrast:

C.R. 25

```python
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip(mode="horizontal", seed=42),
    tf.keras.layers.RandomRotation(factor=0.05, seed=42),
    tf.keras.layers.RandomContrast(factor=0.2, seed=42)
])
```

Next let's load an Xception model, which is pre-trained on ImageNet. We exclude the top of the network by setting `include_top=False`. This excludes the global average pooling layer and the dense output layer. We then add our own global average pooling layer (feeding it the output of the base model), followed by a dense output layer with one unit per class, using the softmax activation function. Finally, we wrap all this in a Keras Model:

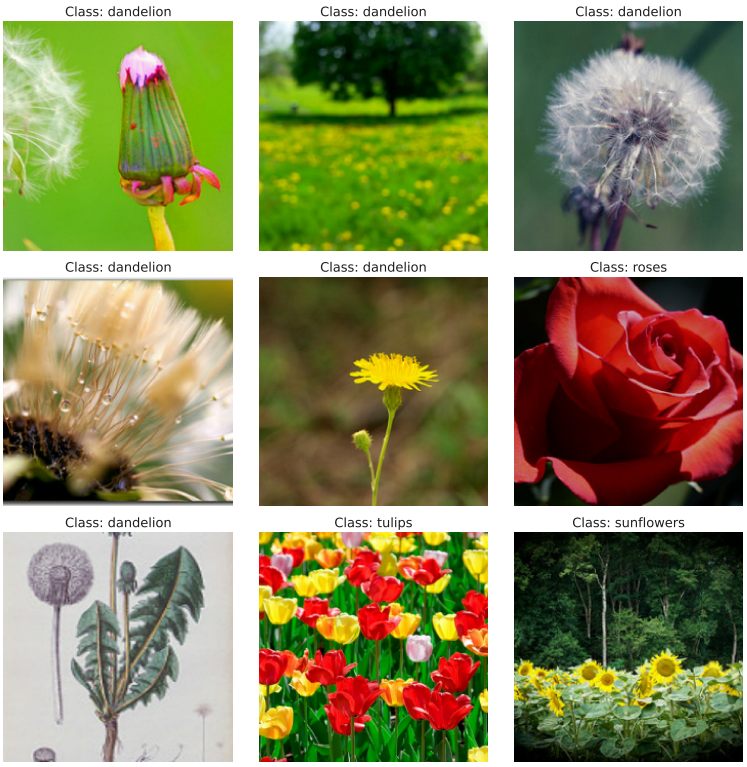


Figure 7.20: Sample images present in the dataset, normalised and all of them have the same dimensions.

C.R. 26
```python
tf.random.set_seed(42)  # extra code  ensures reproducibility
base_model = tf.keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
model = tf.keras.Model(inputs=base_model.input, outputs=output)
```

It's usually a good idea to freeze the weights of the pre-trained layers, at least at the beginning of training[29]:

[29] By not updating the weights of the frozen layers, we avoid tweaking features that are already well-established and generalize well across different tasks.

C.R. 27
```python
for layer in base_model.layers:
    layer.trainable = False
```

Finally, we can compile the model and start training:

C.R. 28
```python
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=3)
```

After training the model for a few epochs, its validation accuracy should reach a bit over 80% and then stop improving. This means that the top layers are now pretty well trained, and we are ready

to unfreeze some of the base model's top layers, then continue training.

For example, let's unfreeze layers 56 and above (that's the start of residual unit 7 out of 14, as you can see if you list the layer names):

C.R. 29
python
```
for layer in base_model.layers[56:]:
    layer.trainable = True
```

Don't forget to compile the model whenever you freeze or unfreeze layers. Also make sure to use a much lower learning rate to avoid damaging the pre-trained weights:

C.R. 30
python
```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10)
```

This model should reach around 92% accuracy on the test set, in just a few minutes of training (with a GPU). If you tune the hyperparameters, lower the learning rate, and train for quite a bit longer, you should be able to reach 95% to 97%.

But there's more to computer vision than just classification. For example, what if you also want to know where the flower is in a picture? Let's look at this now.

C.R. 31
python
```
tf.random.set_seed(42)  # extra code  ensures reproducibility
base_model = tf.keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = tf.keras.layers.Dense(4)(avg)
model = tf.keras.Model(inputs=base_model.input,
                       outputs=[class_output, loc_output])
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)  # added this line
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2],  # depends on what you care most about
              optimizer=optimizer, metrics=["accuracy", "mse"])
```

Glossary

AIC Akaike Information Criterion. 113, 114

ANN Artificial Neural Networks. ix, 119–124, 127, 128, 131, 152

API Application-Programming Interface. 45

BIC Bayesian Information Criterion. 112–114

BMI Body-Mass Index. 19

CART Classification and Regression Tree. 20, 25, 27, 28, 32

CNN Convolutional Neural Networks. ix, x, 98, 148–152, 154, 159, 160, 162, 164, 166, 167, 169–172, 176, 177

CPU Central Processing Unit. 40, 41, 44

DBSCAN Density-Based Spatial Clustering of Applications with Noise. 102, 105

DNN Deep Neural Networks. 128, 130, 151, 152

DT Decision Tree. 19–34

EM Expectation Maximisation. 108, 110

FNN Feedforward Neural Networks. 128

GMM Gaussian Mixture Model. 107

LHS Left Hand Side. 103

LLE Locally Linear Embedding. 60

LLN Law of Large Numbers. 37

LTU Linear Threshold Unit. 124

MAP Maximum a-Posteriori. 114

ML Machine Learning. x, 5, 10, 11, 14, 19–21, 28, 36, 57, 59, 81, 82, 119–121, 125, 156, 168

MLE Maximum Likelihood Estimate. 114

MLP Multi-layer Perceptrons. iv, 119, 127, 128, 130–145

MNIST Modified National Institute of Standards and Technology. 46, 99

MSE Mean Square Error. 32

OOB Out-of Bag. 42, 43

PC Principal Component. 68, 69

PCA Principal Component Analysis. 33, 60, 67–75, 78

PCM Pulse Code Modulation. 85

PDF Portable Document Format. 109, 110, 114

RBF Radial Basis Function. 13, 14, 87

RF Random Forest. vii, 34, 36, 40, 44–46, 55, 57

SAMME Stagewise Additive Modeling using a Multiclass Exponential loss function. 49

SVD Singular Value Decomposition. 68, 69, 73

SVM Support Vector Machines. vii, 5–12, 15–17, 37, 101, 121, 125

TLU Threshold Logic Unit. 124, 125, 127

Bibliography

[1] M. Alaradi and S. Hilal, "Tree-based methods for loan approval," in *2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI)*, IEEE, 2020, pp. 1–6.

[2] P Rajesh, M. Alam, M. Tahernezhadi, C Vikram, and P. Phaneendra, "Real time data science decision tree approach to approve bank loan from lawyer's perspective," in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2020, pp. 921–929.

[3] C. Rodríguez-Pardo et al., "Decision tree learning to predict overweight/obesity based on body mass index and gene polymporphisms," *Gene*, vol. 699, pp. 88–93, 2019.

[4] A. T. Azar and S. M. El-Metwally, "Decision tree classifiers for automated medical diagnosis," *Neural Computing and Applications*, vol. 23, no. 7, pp. 2387–2403, 2013.

[5] J. Mesarić and D. *v*Sebalj, "Decision trees for predicting the academic success of students," *Croatian Operational Research Review*, vol. 7, no. 2, pp. 367–388, 2016.

[6] S. A. Kumar et al., "Efficiency of decision trees in predicting student's academic performance," 2011.

[7] P. K. Dalvi, S. K. Khandge, A. Deomore, A. Bankar, and V. Kanade, "Analysis of customer churn prediction in telecom industry using decision trees and logistic regression," in *2016 symposium on colossal data analysis and networking (CDAN)*, IEEE, 2016, pp. 1–4.

[8] P. Save, P. Tiwarekar, K. N. Jain, and N. Mahyavanshi, "A novel idea for credit card fraud detection using decision tree," *International Journal of Computer Applications*, vol. 161, no. 13, 2017.

[9] Y. Sahin, S. Bulkan, and E. Duman, "A cost-sensitive decision tree approach for fraud detection," *Expert Systems with Applications*, vol. 40, no. 15, pp. 5916–5923, 2013.

[10] W. B. Millard, "The wisdom of crowds, the madness of crowds: Rethinking peer review in the web era," *Annals of Emergency Medicine*, vol. 57, no. 1, A13–A20, 2011.

[11] D Heath, S Kasif, and S Salzberg, "K-dt: A multi-tree learning method," in *Proc. of the Second Int. Workshop on Multistrategy Learning*, 1993, pp. 138–149.

[12] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, IEEE, vol. 1, 1995, pp. 278–282.

[13] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE transactions on pattern analysis and machine intelligence*, vol. 20, no. 8, pp. 832–844, 1998.

[14] Z.-H. Zhou, *Ensemble methods: foundations and algorithms*. CRC press, 2025.

[15] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[16] R. Bellman, "Dynamic programming princeton university press," *Princeton, NJ*, pp. 4–9, 1957.

[17] A. J. Izenman, "Introduction to manifold learning," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 4, no. 5, pp. 439–446, 2012.

[18] R. Nayak, U. C. Pati, and S. K. Das, "A comprehensive review on deep learning-based methods for video anomaly detection," *Image and Vision Computing*, vol. 106, p. 104 078, 2021.

[19] J. Matou *v*sek, "On variants of the johnson–lindenstrauss lemma," *Random Structures & Algorithms*, vol. 33, no. 2, pp. 142–156, 2008.

[20] W. Zhiqiang and L. Jun, "A review of object detection based on convolutional neural network," in *2017 36th Chinese control conference (CCC)*, IEEE, 2017, pp. 11 104–11 109.

[21] T. Kansal, S. Bahuguna, V. Singh, and T. Choudhury, "Customer segmentation using k-means clustering," in *2018 international conference on computational techniques, electronics and mechanical systems (CTEMS)*, IEEE, 2018, pp. 135–139.

[22] R. Kumari, M. Singh, R Jha, N. Singh, et al., "Anomaly detection in network traffic using k-mean clustering," in *2016 3rd international conference on recent advances in information technology (RAIT)*, IEEE, 2016, pp. 387–393.

[23] E. Bair, "Semi-supervised clustering methods," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 5, no. 5, pp. 349–361, 2013.

[24] O. E. Zamir, *Clustering web documents: a phrase-based method for grouping search engine results*. University of Washington, 1999.

[25] S. A. Burney and H. Tariq, "K-means cluster analysis for image segmentation," *International Journal of Computer Applications*, vol. 96, no. 4, 2014.

[26] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[27] E. W. Forgy, "Cluster analysis of multivariate data: Efficiency versus interpretability of classifications," *biometrics*, vol. 21, pp. 768–769, 1965.

[28] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," Stanford, Tech. Rep., 2006.

[29] C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, 2003, pp. 147–153.

[30] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 1177–1178.

[31] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 eighth ieee international conference on data mining*, IEEE, 2008, pp. 413–422.

[32] BBC, *How a kingfisher helped reshape japan's bullet train*, 2019. [Online]. Available: https://www.bbc.com/news/av/science-environment-47673287.

[33] J. F. Vincent, O. A. Bogatyreva, N. R. Bogatyrev, A. Bowyer, and A.-K. Pahl, "Biomimetics: Its practice and theory," *Journal of the Royal Society Interface*, vol. 3, no. 9, pp. 471–482, 2006.

[34] S Agatonovic-Kustrin and R. Beresford, "Basic concepts of artificial neural network (ann) modeling and its application in pharmaceutical research," *Journal of pharmaceutical and biomedical analysis*, vol. 22, no. 5, pp. 717–727, 2000.

[35] S. D. Holcomb, W. K. Porter, S. V. Ault, G. Mao, and J. Wang, "Overview on deepmind and its alphago zero ai," in *Proceedings of the 2018 international conference on big data and education*, 2018, pp. 67–71.

[36] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[37] J. Howe, "Artificial intelligence at edinburgh university: A perspective," *Archived from the original on*, vol. 17, 2007.

[38] I. J. Goodfellow, O. Vinyals, and A. M. Saxe, "Qualitatively characterizing neural network optimization problems," *arXiv preprint arXiv:1412.6544*, 2014.

[39] A. Sebé-Pedrós, "Stepwise emergence of the neuronal gene expression program in early animal evolution," 2023.

[40] D. G. Barrett, A. S. Morcos, and J. H. Macke, "Analyzing biological and artificial neural networks: Challenges with opportunities for synergy?" *Current opinion in neurobiology*, vol. 55, pp. 55–64, 2019.

[41] H.-D. Block, "The perceptron: A model for brain functioning. i," *Reviews of Modern Physics*, vol. 34, no. 1, p. 123, 1962.

[42] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[43] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[44] H. Sompolinsky, "The theory of neural networks: The hebb rule and beyond," in *Heidelberg Colloquium on Glassy Dynamics: Proceedings of a Colloquium on Spin Glasses, Optimization and Neural Networks Held at the University of Heidelberg June 9–13, 1986*, Springer, 2006, pp. 485–527.

[45] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[46] W. Gerstner and W. M. Kistler, "Mathematical formulations of hebbian learning," *Biological cybernetics*, vol. 87, no. 5, pp. 404–415, 2002.

[47] M.-C. Popescu, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis, "Multilayer perceptron and neural networks," *WSEAS Transactions on Circuits and Systems*, vol. 8, no. 7, pp. 579–588, 2009.

[48] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *Ieee Potentials*, vol. 13, no. 4, pp. 27–31, 1994.

[49] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, "Explaining deep neural networks and beyond: A review of methods and applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021.

[50] S. Linnainmaa, "Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden taylor-kehitelmänä," Available in Finnish at https://people.idsia.ch/~juergen/linnainmaa1970thesis.pdf, Master's thesis, University of Helsinki, 1970.

[51] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*, Elsevier, 1992, pp. 65–93.

[52] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[53] H. Knut, "Neural networks p. 7," *University of Applied Sciences Northwestern Switzerland*, 2018.

[54] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[55] G. Hinton et al., "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[56] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[57] Á. Zarándy, C. Rekeczky, P. Szolgay, and L. O. Chua, "Overview of cnn research: 25 years history and the current trends," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2015, pp. 401–404.

[58] Z. Huang and W. Zhao, "Combination of elmo representation and cnn approaches to enhance service discovery," *IEEE Access*, vol. 8, pp. 130 782–130 796, 2020.

[59] Q. Li, X. Li, B. Lee, and J. Kim, "A hybrid cnn-based review helpfulness filtering model for improving e-commerce recommendation service," *Applied Sciences*, vol. 11, no. 18, p. 8613, 2021.

[60] Z. Ouyang, J. Niu, Y. Liu, and M. Guizani, "Deep cnn-based real-time traffic light detector for self-driving vehicles," *IEEE transactions on Mobile Computing*, vol. 19, no. 2, pp. 300–313, 2019.

[61] B. T. Nugraha, S.-F. Su, et al., "Towards self-driving car using convolutional neural network and road lane detector," in *2017 2nd international conference on automation, cognitive science, optics, micro electro-mechanical system, and information technology (ICACOMIT)*, IEEE, 2017, pp. 65–69.

[62] H. Ye, Z. Wu, R.-W. Zhao, X. Wang, Y.-G. Jiang, and X. Xue, "Evaluating two-stream cnn for video classification," in *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, 2015, pp. 435–442.

[63] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, p. 106, 1962.

[64] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[65] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[66] V. Alto, *Data augmentation in deep learning*, 2020. [Online]. Available: https://medium.com/analytics-vidhya/data-augmentation-in-deep-learning-3d7a539f7a28.

[67] G. Deco and E. T. Rolls, "Neurodynamics of biased competition and cooperation for attention: A model with spiking neurons," *Journal of neurophysiology*, vol. 94, no. 1, pp. 295–313, 2005.

[68] M. Zeiler, "Visualizing and understanding convolutional networks," in *European conference on computer vision/arXiv*, vol. 1311, 2014.

[69] C. Szegedy et al., "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[70] X. Xia, C. Xu, and B. Nan, "Inception-v3 for flower classification," in *2017 2nd international conference on image, vision and computing (ICIVC)*, IEEE, 2017, pp. 783–787.

[71] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

[72] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[73] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[74] F. Liu, X. Ren, Z. Zhang, X. Sun, and Y. Zou, "Rethinking skip connection with layer normalization in transformers and resnets," *arXiv preprint arXiv:2105.07205*, 2021.

[75] X. Han et al., "Pre-trained models: Past, present and future," *AI Open*, vol. 2, pp. 225–250, 2021.