SAPIENZA
UNIVERSITÀ DI ROMA

Machine Learning, Notes

Notes from the 2023/2024 Machine Learning (6 CFU/ECTS) course, by Luca Iocchi of Artificial Intelligence & Robotics faculty

If you notice some mistakes:

- email me: *carotenuto.1847282@studenti.uniroma1.it* or
- DM me *@_wisewackywizard_*

Alessandro Carotenuto
ID number

Academic Year 2023/2024

Machine Learning, Notes
Rielaborated Notes. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: carotenuto.1847282@studenti.uniroma1.it

Abstract

This is the rielaborated collection of notes mostly from the course (mainly, the slides) by Luca Iocchi in 2023/2024, part of the Master Degree in Artificial Intelligence and Robotics, plus some integrations from the book "Reinforcement Learning: An Introduction" by Sutton and Barto. Yes, i'm using the **sapthesis** class to format these notes, as if they were an official Sapienza Thesis, 'cause i like it and it's cool as fuck.

At the end of each chapter there is a one-page summary of the most important points and formulas in the chapter, it contains most of the useful bits to understand and review but it's probably not enough by itself.

As a brief overview:

- **Ch.1 - Introduction**: Introduction of Machine Learning and notation.
- **Ch.2 - Evaluation**: How to evaluate the performance of a given hypothesis.
- **Ch.3 - Decision Trees**: Definition, creation and properties of Decision Trees.
- **Ch.4 - Proability Recalls**: One-page summary of proability theory recalls.
- **Ch.5 - Bayesian Learning**: ML problems expressed as probabilistic ones
- **Ch.6 - Prob. models for classification**: Generative and Discriminative
- **Ch.7 - Linear Models for classification**: Least Squares, Perceptron, SVM
- **Ch.8 - Linear Regression**: Predicting Real-number values from the dataset
- **Ch.9 - Kernel Methods**: Simplifying non-linear models
- **Ch.10 - Instance Based Methods**: No formulation of output func, K-NN
- **Ch.11 - Multiple Learners**: Parallel and Sequential Training
- **Ch.12 - Artificial Neural Networks**: Cost func, Backpropag, SGD
- **Ch.13 - Convolutional Neural Networks**: ANN for images
- **Ch.14 - Unsupervised Learning**: Unlabelled or Partially labelled data
- **Ch.15 - Dimensionality Reduction**: Reduction of datasets, PCA
- **Ch.16 - Reinforcement Learning**: Learn from interaction to achieve goals

Contents

Chapter 1

Introduction

Machine Learning is a **Data-Driven approach**, unlike "Planning and Reasoning" and similiar methods, when we use a Machine Learning approach to solve a problem, usually we don't know a perfect model of the environment, so we try to extract it from the data.

We define **Learning** as:

- improving at a **task T**,
- through **experience E**,
- with respect to **performance measure P**.

Sometimes, deterministc solution are better than Machine Learning ones, this could be true in situations where a ML algorithms requires too much time for computation, or too much data is required.

1.1 Target Functions

Extracting a model (a function that describes the system) from the data is usually not an easy task, with functions that often are unfeasible to be computed, so we need to **approximate** those function. Of course, a good approximation would be almost as effective as the real function (used as a model of the environment) but usually the approximated function will not converge to the otpimal function (with the possible exception of really simple and easy Toy-Problems)

Usually, when the learning happens, an error function for the current **state S** of the system is computed, something like:

$$err(S) = V_{train}(S) - \hat{V}(S)$$

Where

- $V_{train}(S)$ is the feedback obtained from the environment in that state.
- $\hat{V}(S)$ is the current estimation of the function.

If we are trying to esteem a parameter, let's say that we call that ω, the update of the estimation will be in a similar form to:

$$\omega \leftarrow \omega + c \cdot err(S)$$

Where **c** is a coefficient that functions as a **learning rate**, usually between 0 and 1.

1.2 Machine Learning problems

Let's define a general machine learning problem as the task of **learning a function** $f : X \rightarrow Y$, where X represents the input space and Y represents the output space. Given a training set D containing information about f, the objective is to approximate this function.

The approximated function is denoted as:

$$\hat{f}(x) \approx f(x) \quad \text{for} \quad x \in \mathbf{X} \backslash X_D$$

And it aims to provide a close approximation of the real f, so the approximated function has to be computed in such a way that maps even instances of **x** that are not in the dataset.

$$X_D = \{x | x \ in \ D\} \subset X \quad with \quad |X_D| <<< |X|$$

It's important to emphasize that $|X_D|$ is considerably smaller than $|X|$, highlighting that the training dataset is a subset of the entire set of possible instances.

We can define different types of Machine Learning, one kind of definition is based on the type of datasets:

- **Supervised Learning**, it involves samples of **input-output** pairs, given in the dataset
- **Unsupervised Learning**, just inputs are given in the dataset, there are no associated output labels. The learning algorithm must uncover patterns, structures, or relationships within the input data without explicit supervision.
- **Reinforcement Learning**, tuples of parameters **(state-action-reward-new state...)** are given in the dataset, the agent learns to make decisions by interacting with an environment. Actions taken by the agent yield rewards and affect the subsequent state, enabling the agent to learn a policy that guides actions to maximize cumulative rewards over time.

1.2.1 Supervised Learning

With respect to the type of function to be learned:

X can be:

- **Discrete**, so made by **finite sets** $(A_1 \times ... \times A_n)$
- **Continous**, so made by a subset of real numbers $\mathcal{R}^n$

Or we can define them based on the type of function to be learned:

Y can be categorized as a:

- **Regression**, $\mathcal{R}^n$
- **Classification**, so with a finite number of outputs $(C_1, ..., C_n)$

Regression
Approximate real-valued functions starting from a dataset.

Classification (Pattern Recognition)
The act of classification returns the class to which a specific instance belongs. A sub-category of pattern recognition is **Concept Learning (binary classification)**.

1.2.2 Unsupervised Learning

Unsupervised learning is a type of machine learning where the model learns patterns and structures in the data without being provided with labeled examples.

- **Clustering**, grouping similar data points together based on certain features or characteristics.
- **Dimensionality reduction**, reducing the number of features in the data while preserving important information.
- **Association**, discovering relationships and patterns within the data.

1.2.3 Reinforcement Learning

Reinforcement Learning (RL) entails the learning of a decision-making policy for an agent, mapping states to actions. Unlike supervised learning, RL lacks direct supervision, relying on sparse and delayed rewards from the environment to guide learning. The goal is to develop a policy that maximizes cumulative rewards by effectively exploring and exploiting the environment, even in the absence of explicit output guidance.
In synthesis in RL the agent is dedicated to:

- Learning a policy (state-action function)
- No supervised output available, only sparse and time-delayed rewards.

1.3 Notation and Examples

Data set:

- Input: X
- Output: Y

Target Function: $c : X \to Y$

Notation

• X	: instance space
• $x \in X$	: one instance
• $D = \{(x_i, c(x_i))_{i=1}^n\}$	: data set
• $c(x)$	: value of the target function over x
• H	: hypothesis space
• $h \in H$	: one hypothesis (approximation of c)
• $h(x)$	: estimation over x

We **assume the dataset to be without noise**. Every **hypothesis** $h \in H$ is of course, a function, while x is the input. The input type depends on the probelm and the hypothesis depends on the hypothesis space that we are defining.

Given a training set D, find the best approximation $h^* \in H$ of the target function c.

Solving this kind of problems requires three steps:

- Define the hypothesis space H
- Define a performance metric to determine the *best* approximation
- Define an appropriate algorithm

We can model the problem as a *searching problem*, given H the space of possible hypothesis, we have to find the **best one** among them (according to the performance metric we defined)

Even if $h(x)$ can be computed for every $x \in X$ we have that $h(x_i) = c(x_i)$ can be verified only for istances that appear in the data set.

We define an hypothesis as ***consistent*** *when $h(x) = c(x)$ for each training example (x,c(x)) in D.*

But the challenge is to find a $h(x)$ that predicts correct values for instances ouside of D, such as $h(x') = c(x')$ where $x' \notin D$. To do that we need a way of measuring performance, we define a **perfomance measure** a method based on evaluating $c(x_i) = h(x_i)$ for all $x_i \in D$.

1.3.1 Version Spaces

The version space, $VS_{H,D} \equiv \{h \in H, |\text{Consistent}(h, D)\}$ is the subset of hypotheses from H consistent with all training examples in D. These are all candidates to be the optimal hypothesis.

The simplest "algorithm" to produce a Vector Space consists of

- List every single hypothesis in H.
- For Each *training example* $(x, c(x))$, remove those hypothesis which $h(x) \neq c(x)$
- Output the list of hypothesis

Representation vs Generalization

It is believed that any hypothesis that approximates the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples. However, if the hypothesis space H is "too powerful" so, with too much **representation** power, the hypothesis that we can find could also be **too tailored to the dataset** with the risk of losing **generalization** potential.

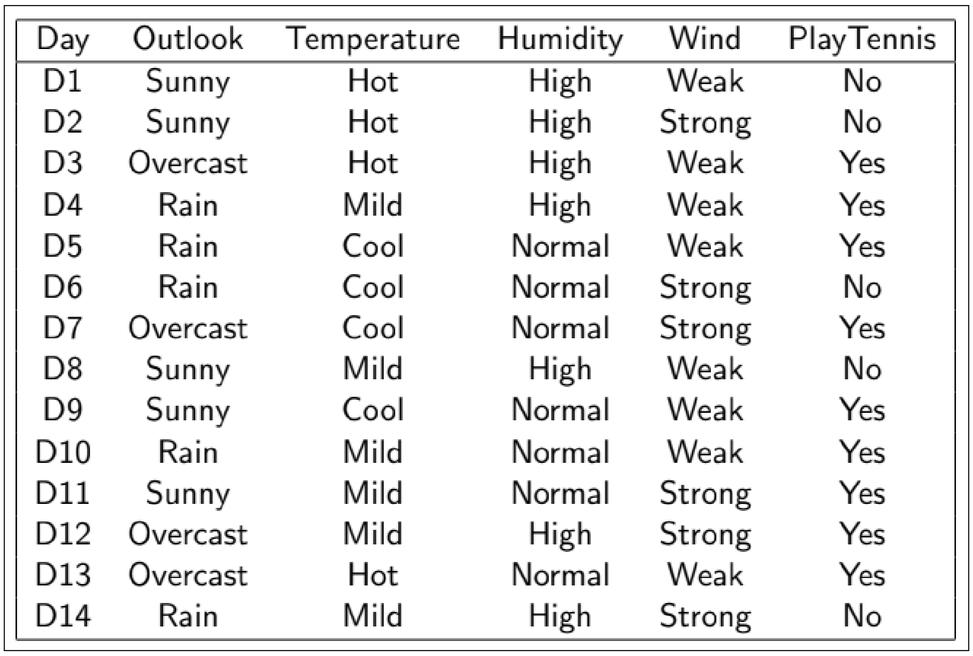
Typically, in order to address this challenge, we deliberately **introduce certain biases** into the selection of our hypothesis space. While this deliberate biasing may result in a loss of some representation power, it concurrently enhances our ability to generalize effectively. Striking the right balance in choosing the hypothesis space **H** is crucial to achieve optimal outcomes in our analytical pursuits.

When we talk about **introducing biases** into the hypothesis space, we essentially imply that **we limit the range of potential hypotheses or models that we consider**. This constraint is intentional and serves a specific purpose: to streamline our search for the most probable and relevant hypotheses. By doing so, we sacrifice some degree of inclusiveness and representation in the space of possible hypotheses. Consequently, our model might not capture every nuance or variation in the data, but it enables more robust and accurate generalizations.

However, it is essential to recognize that an excessive bias towards a limited hypothesis space can be detrimental. It may lead to oversimplification and overlooking crucial complexities in the underlying data. On the other end of the spectrum, an overly expansive hypothesis space can result in overfitting, where the model becomes excessively complex and starts fitting noise instead of genuine patterns. Striking the right balance is essential to avoid both underfitting and overfitting.

1.3.2 Concept Learning Example: PlayTennis

We present an example of Concept Learning, we have a **discrete** input **X** and a discrete, binary output **Y** which is Yes/No. The context in this case is creating a model that given some attributes of the day, tells us wether we can or cannot play Tennis that day. We have the dataset **D**:



Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Figure 1.1. Dataset for the Tennis Example

We can choose an hypothesis space in the following (arbitrary) form:

$$h_k = < b_1, b_2, b_3, b_4 >$$

$$h_k = \text{YES, if } b_k = a_k \text{ for } k \in (1, 4)$$

Where

- a_k is the k-sim attribute of the problem.
- b_k is the k-sim attribute considered by the hypothesis.

What does this mean? That means that we are considering a h defined by a conjunction of contraints, for example: The function $h_1 = < Sunny, Hot, High, Strong >$ returns **YES** if, given the four attributes, they match with the attributes of the function, so, looking at the table, if x_d is the set of attributes of day number d, we have that

$$h_1(x_1) = NO$$

Because x_1 = Sunny, Hot, High, Weak and confronting with the h_1 function, we have: Sunny=Sunny, Hot=Hot, High=High but Strong≠Weak

This is just a single hypothesis from an **ad hoc** hypothesis space that we arbitrarly decided to use. We'll came back to this example problem later.

1.4 Introduction: Summary

Machine Learning is a **data-driven approach** , it emphasizes the reliance on data to construct models when a perfect model of the environment is unknown. The concept of learning is defined as **enhancing task performance** based on **experience** and measured against a **performance criterion**.

Machine learning models try to approximate a **target function** , usually using an error function $err(S) = V_{train}(S) - \hat{V}(S)$. If a parameter is being estimated, the update rule will be in a similar form to $\omega =\leftarrow \omega + c \cdot err(S)$
Different types of Machine Learning problems include:

- **Supervised Learning**, it involves samples of **input-output** pairs, given in the dataset. Output of Supervised Learning can be real-value functions (**regression**) or a finite number of output (**classification**)
- **Unsupervised Learning**, just inputs are given in the dataset, ere are no associated output labels. The learning algorithm must uncover patterns, structures, or relationships within the input data without explicit supervision, the major task are clustering, dimensionality reduction and association.
- **Reinforcement Learning**, tuples of parameters **(state-action-reward-new state...)** are given in the dataset, the agent learns to make decisions by interacting with an environment. Actions taken by the agent yield rewards and affect the subsequent state, enabling the agent to learn a policy that guides actions to maximize cumulative rewards over time.

Recurrent Notation

• X	: instance space
• $x \in X$	: one instance
• $D = \{(x_i, c(x_i))_{i=1}^{n}\}$	: data set
• $c(x)$	: value of the target function over x
• H	: hypothesis space
• $h \in H$	: one hypothesis (approximation of c)
• $h(x)$	: estimation over x

We define an hypothesis as **consistent** when $h(x) = c(x)$ for each training example (x,c(x)) in **D**. The subset of hypothesis from H consistent with all training examples in **D** is called the **version space** of **H**.

Balancing hypothesis space is critical for effective **generalization** ; a well-approximated target function over training examples should extend to unobserved ones, and if the representational power of **H** is too high, that does not happen. Intentional bias in hypothesis selection improves generalization, but excess bias can oversimplify or miss important complexities. Striking the right balance in hypothesis space is key to prevent **underfitting** and **overfitting**, ensuring optimal analytical outcomes.

Chapter 2

Evaluation for Classification

To evaluate the performance of a given hypothesis h, we proceed in terms of *accuracy* and *error rate.*

Let's consider a tipical classfication problem, given:

- **D** the Dataset, a probability distribution over the entire **X** (all possible inputs)
- **S** sample of n instances from **X** according to **D** for which we know $f(x)$

2.1 Errors and Estimators

Let's now define two kind of error

- **True error,** the probability to missclassify an instance drawn at random according to **D**
$$err_D(h) = P[f(x) \neq h(x)] \quad x \in \mathbf{D}$$
- **Sample error,** is the relative frequency of the missclassification in the sample, basically is
$$\frac{n_{miss}}{n}$$

*The **sample error** is computed on a small portion of the data, it approximates the **true error** which cannot be computed.*

If we define the **bias** as $bias = E_s[err_s(h)] - err_d(h)$, we will need an independent choice of S and h to have $E_s[err_s(h)] = err_d(h)$ and consequent bias $= 0$, so we consider $err_s(h)$ an **unbiased** estimator.

2.1.1 Confidence Interval for $err_D(h)$

We can define a **confidence interval** for where $err_D(h)$ lies with respect to $err_S(h)$: With N% probability $err_D(h)$ lies in the interval

$$err_S(h) \pm z_n \sqrt{\frac{err_S(h)(1 - err_S(h))}{n}}$$

Where

- $\sqrt{\frac{err_S(h)(1-err_S(h))}{n}}$ is the standard error of a proportion, basically, a statistic indicating how greatly a particular sample proportion is likely to differ from the proportion if extended to the whole population, the formula is usually $\sqrt{\frac{p(1-p)}{n}}$ but our p is $err_S(h)$
- z_n is the is the critical value from the standard normal distribution corresponding to the desired confidence level. According to the normal distribution, we choose a % of confidence for our interval definition and then we look at the corrisponding z_n number, some example of corrispondences are:

N%	50%	80%	90%	95%	99%
z_n	0.67	1.28	1.64	1.96	2.58

2.1.2 Computing unbiased estimators

To compute an unbiased estimator we need to

- Make a partition of our Dataset
  - $D = T \cup S$, Train and Sample set
  - Disjointed Sets
  - As a rule of thumb for medium size Dataset, Training set should be around 2/3 of the entire Dataset.
- Compute h using training set T.
- Evaluate **sample error** n_{miss}/n for h

Of course, if we use a greater fraction of D as a sample, we will have a ***better estimation*** *of $err_D(h)$, but with less training data, our model will not perform as good as having a greater training set.*

We consider h as the solution of the L algorithm on the T training set, so:

$$h = L(T)$$

We can compute $E_S[err_S(h)]$ can be approximated by averaging the **sample error** computed on a series of **samples**, an algorithm to do that is the **K-Fold cross validation** .

Algorithm 1 K-Fold Cross Validation

```
Require: $S_1, S_2, ..., S_k$ partitions of the dataset $D$          ▷ $k \geq 30$ suggested
    for $i = 1, ..., k$ do                                             ▷ For each sample
        $T_i \leftarrow \{D - S_i\}$        ▷ Use every partition besides $S_i$ for the training set
        $h_i \leftarrow L(T_i)$                          ▷ Compute the hypothesis $h$ using $T_i$
        $\delta_i \leftarrow err_{S_i}(h_i)$       ▷ Use $S_i$ as test set to compute the sample error
    end for
    return $err_{L,D} = \frac{1}{k} \sum_{i=1}^{k} \delta_i$
```

*We define the **accuracy** as $acc_{L,D} = 1 - err_{L,D}$*

*The same algorithm can be used for **regression** problems, it's important to **change the metric** for evaluating the error, maybe using a **mean absolute error (MAE)***

2.2 Performance Metric

While **accuracy** can be an insightful indicator, it might be not good enough by itself as a performance metric. It turns out it is useful only with balanced datasets.

Example of Binary classification, if h_1 always returns '+' and h_2 returns the correct classification 80% of the time. If the dataset contains more than 80% of '+' h_1 will be considered more accurate, which is false.ov

True and False, Positive and Negative

Let's define the following concepts

- **Positive/Negative**, we can call Positive/Negative the result of the classification using our hypothesis h
- **True/False**, we can call True/False the matching of our classification with the real class of the input.

We can have them in **every combination** (4)

Some performance metric usable in classification are

• Recall:	True positives / Real Positives
• Precision:	True positives / Predicted Positives

A way of visualizing all the **missclassification** is plotting a multi-class confusion matrix, mapping every class with how many times it was missclassified in every other class.

2.3 Evaluation for classification: Summary

To evaluate the performance of a given hypothesis h, we proceed in terms of *accuracy* and *error rate.*

Let's now define two kind of error:

- **True error,** the probability to missclassify an instance drawn at random according to **D** $err_D(h) = P[f(x) \neq h(x)] \quad x \in \mathbf{D}$
- **Sample error,** is the relative frequency of the missclassification in the sample, basically is n_{miss}/n

The sample error is computed on a small portion of the data, it approximate the true error *which cannot be computed*, but we can define a **confidence interval** for where $err_D(h)$ lies w.r.t $err_S(h) = err_S(h) \pm z_n\sqrt{\frac{err_S(h)(1-err_S(h))}{n}}$.

To compute an **unbiased estimator** we need to

- Make a partition of our Dataset (of Train and Sample set, disjointed)
- Compute h using training set T
- Evaluate **sample error** n_{miss}/n for h

We can then approximate $E_S[err_S(h) = err_D(h)]$ can be approximated by averaging the **sample error** computed on a series of **samples**, an algorithm to do that is the **K-Fold cross validation** .

K-Fold Cross Validation: *Splits data into 'K' subsets, train 'K' models, each using K-1 subsets for training and 1 for validation. Repeat until each subset is used for validation. Assess model performance. Can also be used for **regression** problems, it's important to change the metricfor evaluating the error, maybe using a **mean absolute error** .*

We define the **accuracy** of a model L as $acc_{L,D} = 1 - err_{L,D}$, it can be an insightful indicator, it might be not good enough by itself as a performance metric.

Some performance metric usable in classification are:

• Recall:	True positives / Real Positives
• Precision:	True positives / Predicted Positives

A way of visualizing all the **missclassification** is plotting a multi-class confusion matrix, mapping every class with how many times it was missclassified in every other class.

Chapter 3

Decision Trees

In some types of problems, we can choose the hypothesis space of the *set of all Decision Trees*, for example in **concept learning.**

Decision Trees, definition

A decision tree is a tree with the following characteristics
- Each internal **node** represents an attribute
- Each individual **branch** represent a value of an attribute
- Each **leaf** represents an **output**

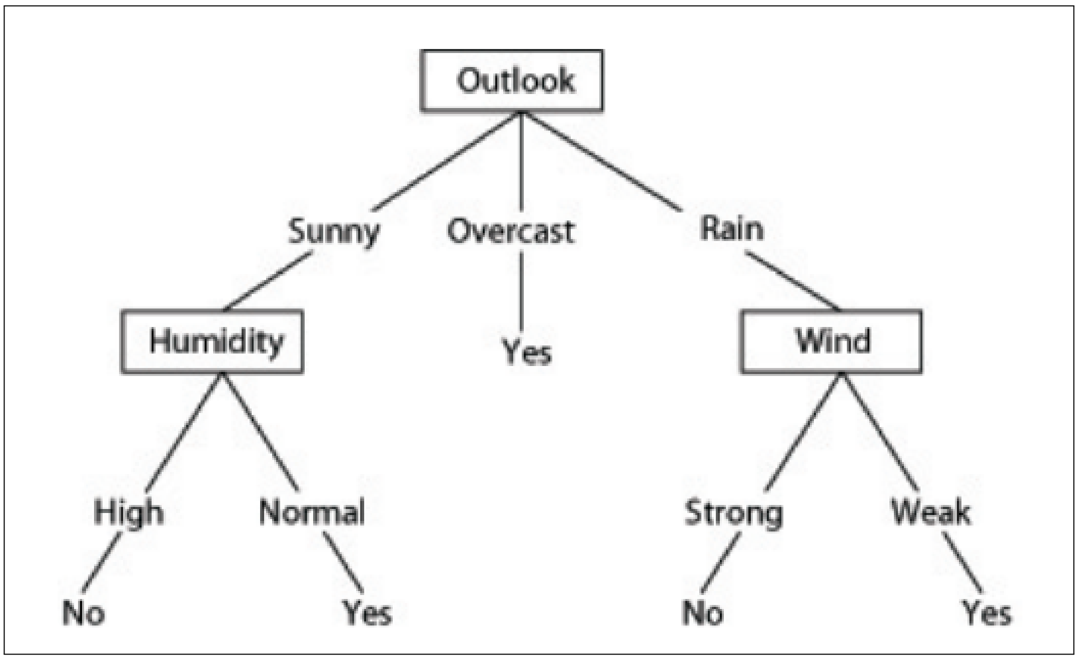


Figure 3.1. A simple Decision Tree for the PlayTennis example in Chapter 1

3.1 Alternative Forms

3.1.1 Logical Conjunctions

A tree can be represented in a series of disjunction of logical statements.

- Each statement will be the **conjunction** of all possible attributes value that lead to a **positive result** (A path of "And" all the way to the **leaf**).
- The full representation will be the **disjunction** of every statement

With respect to the Figure 3.1:

- A path to a leaf is, for example:

$$\text{Outlook} = \text{Sunny} \wedge \text{Humidity} = \text{Normal}$$

- Another path is easily:

$$\text{Outlook} = \text{Rain} \wedge \text{Wind} = \text{Weak}$$

- The last one is:

$$\text{Outlook} = \text{Overcast}$$

The logical representation is the **disjunction** of all those statements (Statement 1 $\vee$ Statement 2 ...)

3.1.2 Rules Form

A simpler version, writing every path to a leaf and expliciting its output, for example:

$$\text{IF Outlook} = \text{Sunny} \wedge \text{Humidity} = \text{Normal}$$
$$\text{THEN Output} = \text{YES}$$
$$\ldots$$

3.2 Creating a Decision Tree

3.2.1 ID3 Algorithm

Algorithm 2 ID3 Algorithm

```
Require: $E_1, ..., E_n$ examples, $A_1, ...A_k$ attributes, $T$ target attribute

  Create Root Node                                                    ▷ Setup Phase
  if All $E$ are positives then return Root node labeled with +
  end if
  if All $E$ are negatives then return Root node labeled with -
  end if

  if No attributes are given then return Root node labeled with
    the most common value of the $T$ target attribute
  else
      Choose $A$ as a decision attribute for the Root
      for each value $v_i$ of $A$ do
          Add a new branch from Root corresponding to the test $A = v_i$
          Consider $E_{v_i}$ as a subset of the Examples that have $A = v_i$
          if $E_{v_i}$ is empty then add a leaf node labeled with the
                most common value of the $T$ target attribute
          else Add the tree ID3($E_{v_i}$,$T$,Attributes - $\{A\}$)
          end if
      end for
  end if
```

Now the problem is, how do we decide the **best attribute** to consider as a decision attribute for the **Root**

3.2.2 Information Gain Criteria

We need a statistic that determines how well a given attribute separates the training examples according to their target classification, we call that **information gain** and it's the criteria used by the **ID3 Algorithm**, we define the **information gain** of an attribute as a **reduction of entropy** , a measure of the **impurity** of the **set**.

Entropy

For a **concept learning** problem we define

$$\textbf{Entropy}(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

Where:

- p_+ is the proportion of **positive** examples in S
- p_- is the proportion of **negative** examples in S

We can, of course, extend the concept of **entropy** to a general classification problem with **c** classes.

$$\textbf{Entropy}(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

In **concept learning** the entropy is max when the relative frequency of the two labels are equal

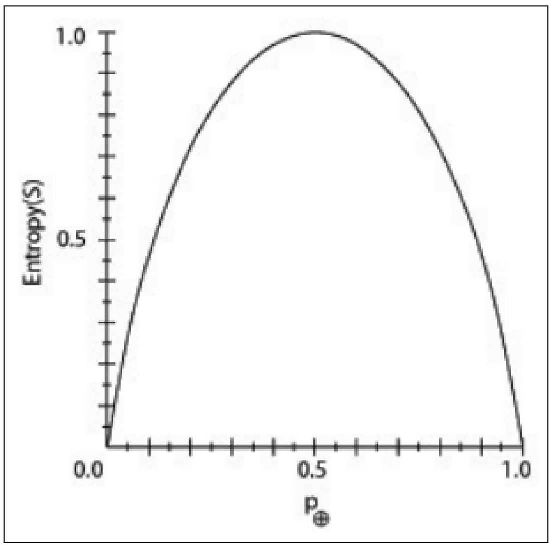


Figure 3.2. Entropy variation in concept learning problems

So, the **expected reduction** in entropy of the sample S caused by knowing the value of attribute A is:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

Where

- Values(A) is the **set of all possible values of** A.
- S_v is the **subset of** S having v as the value for A.
- $|S|$ is the **cardinality** of the set, the number of elements it's composed of

So, basically, **for every value that** A **can assume**, the entropy of the **subset** multiplied by its rapport on the total set, is subtracted from the entropy of the total set. Actually, the formula above is the **residual entropy** after the expected reduction, and since the higher residual entropy, the better the separation of the dataset by the attribute, we prefer attributes with **lower expected reduction** in entropy, hence, higher **gain** .

3.2.3 Other Properties of Decision Tree and ID3

*The **hypothesis space** in ID3 includes all possible decision trees that can be constructed from the given features and their possible values. The algorithm explores and searches through this hypothesis space to find the decision tree that best fits the training data and accurately predicts the target variable, we consider the hypothesis Space of all possible Decision Trees by ID3 as **complete**.*

Issues on Decision Tree Learning

- Determining how deeply to grow the DT
- Handling continuous attributes
- Choosing appropriate attribute selection measures
- Handling training data with missing attribute values
- Handling attributes with different costs

Overfitting and Pruning

To prevent overfitting in decision tree models, it's essential to stop tree growth when data splits lack **statistical significance**. Begin with a full tree and employ **post-pruning** to trim unnecessary branches. Decide the appropriate tree size using a separate dataset **for evaluation and statistical tests** for accuracy estimation. Also, consider the complexity of encoding examples and the tree itself to strike a balance between capturing patterns and avoiding excessive intricacies. These steps ensure the model generalizes well and avoids overfitting.

Reduced-Error Pruning

To avoid overfitting in the context of decision trees, the following approach is commonly employed:

- **Data Splitting**: Start by dividing the available data into training and validation sets.
- **Pruning Process**:
  - **Iterative Pruning**: Proceed iteratively until further pruning harms accuracy.
  - Node **Evaluation**: Evaluate the impact of pruning each potential node by temporarily removing its subtree and assigning the most frequent classification within that subtree.
  - **Greedy Selection**: Greedily choose the node whose removal leads to the greatest improvement in validation set accuracy.

Rule Post-Pruning

The **Rule Post-Pruning method** involves:

- Constructing an initial decision tree
- Converting it into a set of rules
- Independently refining each rule to prevent overfitting.
- The refined rules are then organized into a preferred sequence for efficient utilization.

When dealing with **missing values** of variable A in training examples within a decision tree, the approach involves utilizing available information for classification. By assigning **common values of A at respective nodes and calculating probabilities for possible values**, the process ensures a consistent classification methodology. This method is then applied to classify new examples, **maintaining uniformity in handling missing data**.

Formal Definition of Overfitting

For a more formal definition of overfitting we can state that when the **sample error** for an hypotesis h_1 is lower than the sample error for another hypotesis h_2 **despite** the **true error** of h_1 being bigger, we have **overfitting** (our hypotesis it's too much representative of the dataset)

3.3 Decision Trees: Summary

In some types of problems, we can choose the hypothesis space of the *set of all Decision Trees*, which is a **tree** with the following characteristics:

- Each internal **node** represents an attribute
- Each individual **branch** represent a value of an attribute
- Each **leaf** represents an **output**

Alternative Decision Trees Forms

- **Logical Conjunctions**. this method interprets a decision tree using logical statements, presenting attribute value combinations (**AND**) guiding outcomes, integrated into a disjunction (**OR**) for a comprehensive view.
- **Rules form**. s simpler version, writing every path to a leaf and expliciting its output in a **IF-THEN** chain.

To create a Decision Tree, we can use the **ID3 Algorithm** , it utilizes recursion to iteratively select the best attribute for data splitting based on **information gain** . This process creates decision tree nodes, each representing an attribute. The algorithm repeats this recursive process for each subset of data until the data is fully classified or no more attributes are available for splitting, *resulting in a complete decision tree.*

Information gain in ID3 measures an attribute's ability to segregate training examples by their target classification. It's computed as a reduction in **entropy** representing set impurity. In **concept learning**:

- **Entropy**(S) $= -p_+ \log_2 p_+ - p_- \log_2 p_-$
- **Gain** $(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \dfrac{|S_v|}{|S|} \text{Entropy}(S_v)$

Where:

- p_+ is the proportion of **positive** examples in S
- p_- is the proportion of **negative** examples in S
- $\text{Values}(A)$ is the **set of all possible values of** A.
- S_v is the **subset of** S having v as the value for A.
- $|S|$ is the **cardinality** of the set, the number of elements it's composed of

Since the higher residual entropy, the better the separation of the dataset by the attribute, we prefer attributes with **lower expected reduction** in entropy, hence, higher **gain** .

To avert **overfitting** in decision trees, halt growth at insignificant data splits. Start with a complete tree, then **post-prune** to refine size using a distinct dataset and statistical tests. Factor in complexity for effective pattern capture, striking a balance. These actions enhance model generalization, preventing overfitting. **Reduced-Error Pruning** or **Rule Post-Pruning** are methods to reduce overfitting.

Chapter 4

Probability Recalls

We use probability to measure **uncertainty** in our system. It's a **function** that maps an **event** *(a subset of the set of all possibilities Ω)* to $x \in \mathbb{R}$ between 0 and 1.

Random Variables

Random variables, are **functions** consdidering tha they are mapping values from Ω to a **certain range** in $\mathbb{R}$, but are also considered **variables** since they can assume a specific value in that range.

Some definitions

- A **proposition** is an event with an assignment of a **random variable**
- A probability **distribution** is a function assigning a probability value to all possible assignments of a random variable.
- A **Joint probability distribution** for a set of random variables gives the probability of every atomic joint event on those random variables.
- The **conditional prob.** of events a and b is $P(a|b) = \dfrac{P(a \wedge b)}{P(b)}$ if $P(b) \neq 0$
  - **Product Rule**, $P(a|b)P(b) = P(b|a)P(a)$
- Two variables are **independent** if $P(a|b) = P(a)P(b)$
- X and Y are **conditionally independent** given Z if

$$P(X \vee Y|Z) = P(X|Y \vee Z)P(Z) = P(X|Z)P(Y|Z)$$

- The **Bayes Rule** , $P(a|b) = \dfrac{P(b|a)P(a)}{P(b)}$

Bayesian Networks

A Bayesian Network is a graphical notation for conditional independence assertions and hence for compact specification of full joint distributions.

- **Set of nodes**, one for variable
- **Directed**, acyclic graph of direct influences
- **Conditional Distribution**, for each node given its parents: $P(X_i|Parents(X_i))$ (condit**ional probability table**)

Chapter 5

Bayesian Learning

Bayesian methods serve two roles:

- **Practical learning algorithms** (e.g., Naive Bayes) for probability-based hypothesis correctness and probabilistic predictions, incorporating prior knowledge and observed data.
- Valuable **conceptual framework** and a standard for evaluating other learning algorithms by combining prior probabilities with data.

Given a target function $f(x) : X \to V$, dataset $\mathbf{D}$ and new instance x' we define the **best prediction** $\hat{f}(x') = v^*$ as

$$v^* = \mathbf{argmax}_{v \in V} P(v|x', D)$$

Probability distribution

We can express with $P(H|D)$ the probability distribution of the hypothesis space given the dataset

5.1 Maximum a Posteriori hypothesis

We define the **MAP** , Maximum a Posteriori hypothesis as:

$$h_{\text{MAP}} = \mathbf{argmax}_{h \in H} P(h|D)$$

$$h_{\text{MAP}} = \mathbf{argmax}_{h \in H} \frac{P(D|h)P(h)}{P(D)} = \mathbf{argmax}_{h \in H} P(D|h)P(h)$$

In this case, let's consider the probabilities of the **random variable** we are considering:

- $P(h)$ =prior probability of hypothesis h, so **without a dataset**

- $P(D)$ = prior probability of training data D, it is just a **normalization factor** (in *argmax* problems usually does not need to be computed)

That we can simplify considering **every hypothesis** to have the same **probability** we can choose the **Maximum Likelihood** probability (we expand the **conditional probability** and the $P(h)$ is simplified, the rest does not contain **h** so has not influence on the **argmax**)

$$h_{\text{ML}} = \mathbf{argmax}_{h\in H} P(D|h)$$

The ***brute force*** *approach to calculation is to calculate every instance of $P(h|D)$ and take the* ***maximizing*** *h.*

5.2 Bayes optimal classifier

The maximizing hypothesis for the **dataset** is the most probable classification hypothesis for the dataset itself, this **does not mean that will be the most probable for every new entry in the dataset**. The **Bayesian optimal classifier** aims to minimize the probability of misclassification, and in theory, it achieves the lowest possible error rate given the prior probabilities and the true underlying data distribution. It represents the ideal classification method when we have complete and perfect knowledge about the underlying probability distributions of the data and their relationships. Considering a new instance $x \notin D$

$$P(v_j|x, D) = \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D)$$

To put that in a discoursive manner, the **probability** of the result being a certain class, given the Dataset and the new instance, is the sum of the **probability** of that class given the new instance and every single **hypothesis**, weighted by its **posterior probability** (probability of hypothesis given the dataset).

So, to obtain the **class** v_j of a new instance x:

Example

$$\begin{aligned} P(h_1|D) = 0.4, \quad P(\ominus|x, h_1) = 0, \quad P(\oplus|x, h_1) = 1 \\ P(h_2|D) = 0.3, \quad P(\ominus|x, h_2) = 1, \quad P(\oplus|x, h_2) = 0 \\ P(h_3|D) = 0.3, \quad P(\ominus|x, h_3) = 1, \quad P(\oplus|x, h_3) = 0 \end{aligned}$$

Therefore,

$$\sum_{h_i \in H} P(\oplus|x, h_i)P(h_i|D) = 0.4$$

$$\sum_{h_i \in H} P(\ominus|x, h_i)P(h_i|D) = 0.6$$

and so $v_{OB} = \arg\max_{v_j \in V} \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D) = \ominus$

Binomial and Multinomial Distribution

The binomial distribution can be described as a probability model that calculates the likelihood of having a specific number of successes (k) in a fixed number of independent, identical trials (n), each with the same probability of success (p). The probability of obtaining exactly k successes is given by the formula:

$$P(X = k) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

In this formula:

- $P(X = k)$ is the probability of obtaining exactly k successes.
- n represents the total number of trials.
- k is the number of successes you want to find the probability for.
- p is the probability of success in a single trial.
- $\binom{n}{k}$ is the binomial coefficient, calculated as $\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!}$, with "!" representing the factorial function.

If we have **more** than two possible outcomes for the experiment, we have a **multinomial** distribution, the probability mass function for the multinomial distribution is given by the formula:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_d = x_d) = \frac{n!}{x_1! \cdot x_2! \cdot \ldots \cdot x_d!} \cdot p_1^{x_1} \cdot p_2^{x_2} \cdot \ldots \cdot p_d^{x_d}$$

where:

- $P(X_1 = x_1, X_2 = x_2, \ldots, X_d = x_d)$ is the probability of observing x_1 occurrences of outcome 1, x_2 occurrences of outcome 2, and so on, in n trials.
- n is the total number of trials.
- d is the number of different outcomes.
- $x_1, x_2, \ldots, x_d$ represent the number of occurrences for each outcome, with $\sum_{i=1}^{d} x_i = n$.
- $p_1, p_2, \ldots, p_k$ are the probabilities of each outcome occurring in a single trial, with $\sum_{i=1}^{d} p_i = 1$.

5.3 Naive Bayes Classifier

The **Naive Bayes Classifier** is a machine learning algorithm that approximates the solution by utilizing conditional independence. Specifically, it assumes that X is conditionally independent of Y given Z, which can be expressed as:

$$P(X, Y|Z) = P(X|Y, Z) \cdot P(Y|Z) = P(X|Z) \cdot P(Y|Z)$$

In the context of the Naive Bayes Classifier for a target function $f : X \to V$, where instances are described by attributes $a_1, a_2, \ldots, a_n$, it calculates the most probable value of $f(x)$ without explicitly representing hypotheses. This is done using the following expression:

$$\text{argmax}_{v_j \in V} P(v_j|x, D) = \text{argmax}_{v_j \in V} P(v_j|a_1, a_2, \ldots, a_n, D)$$

Given a dataset D and a new instance $x = a_1, a_2, \ldots, a_n$, the most probable value of $f(x)$ is determined as:

$$v_{\text{MAP}} = \text{argmax}_{v_j \in V} P(v_j|a_1, a_2, \ldots, a_n, D) = \text{argmax}_{v_j \in V} \frac{P(a_1, a_2, \ldots, a_n|v_j, D) \cdot P(v_j|D)}{P(a_1, a_2, \ldots, a_n|D)}$$

This expression makes use of **Bayes' rule**, the **Naive Bayes assumption** states that:

$$P(a_1, a_2, \ldots, a_n|v_j, D) = \prod_i P(a_i|v_j, D)$$

The Naive Bayes classifier assigns a class to a new instance x by considering:

$$v_{\text{NB}} = \text{argmax}_{v_j \in V} P(v_j|D) \prod_i P(a_i|v_j, D)$$

In this classifier, the class of the new instance x is determined by evaluating the conditional probabilities of each attribute a_i given each possible class v_j, and then multiplying these probabilities with the prior probabilities of each class $P(v_j|D)$ before taking the **argmax** to make the final classification decision.

Algorithm 3 Naive Bayes Learn

```
Require: Target function $f : X \to V$, $X = A_1 \times \ldots \times A_n$, $V = \{v_1, \ldots, v_k\}$
  Data set $D$, new instance $x = \langle a_1, a_2, \ldots, a_n \rangle$
  for each target value $v_j \in V$ do
      $\hat{P}(v_j|D) \leftarrow$ estimate $P(v_j|D)$
      for each attribute $A_k$ do
          for each attribute value $a_i \in A_k$ do
              $\hat{P}(a_i|v_j, D) \leftarrow$ estimate $P(a_i|v_j, D)$
          end for
      end for
  end for
  $v_{NB} \leftarrow \arg\max_{v_j \in V} \hat{P}(v_j|D)$
  return $v_{NB}$
```

Naive Bayes is often preferred to the Optimal Bayes given the unfeasibility of this last one, that is due to the hypotesis space beaing too large.

5.3.1 Naive Bayes Classification for Text

Learning Phase and Classification Phase

In the context of text classification, we have a set of documents already classified into different classes, such as spam or not spam, or positive/negative sentiment. The goal is to learn a function that can predict the class of new, unlabeled documents.

In the classification phase, we take a new, unlabeled document and want to determine which class it belongs to. We calculate the probability of it belonging to each class and then choose the class with the highest probability as the predicted class for the document.

Naive Bayes Assumption

The Naive Bayes assumption is that the presence or absence of words in a document is independent of each other given the class of the document. It can be written as:

$$P(\text{document}|\text{class}, D) = \prod_{i=1}^{n} P(\text{word}_i|\text{class}, D)$$

We calculate the probability that a particular word appears in a document of a specific class:

$$P(\text{word}_i|\text{class}, D)$$

To apply Naive Bayes to text classification, we often use a "**bag of words**" representation. This means that we don't consider the order of words in the document; we only care about their presence or frequency.

Approaches

There are two common ways to represent a document **as a feature vector**:

- **Boolean Feature Vector**, each element in the vector is a binary value (1 or 0) representing whether a specific word is present in the document. This corresponds to a Multivariate Bernoulli distribution.

$$P(\text{document}|\text{class}, D) = \prod_{i=1}^{n} [P(\text{word}_i|\text{class}, D)]^{d_i} [1 - P(\text{word}_i|\text{class}, D)]^{(1-d_i)}$$

- **Ordinal Feature Vector**, each element in the vector represents the count of how many times a specific word appears in the document. This corresponds to a Multinomial distribution.

$$P(\text{document}|\text{class}, D) = \frac{n!}{d_1! \cdot d_2! \cdot \ldots \cdot d_n!} \prod_{i=1}^{n} [P(\text{word}_i|\text{class}, D)]^{d_i}$$

For the Multivariate Bernoulli model, we calculate the probabilities using **maximum-likelihood estimation**. This involves **counting how often each word occurs in documents of a specific class**.

$$P(\text{word}_i|\text{class}, D) = \frac{t_{i,j} + 1}{t_j + 2}$$

- $t_{i,j}$: number of documents in D of class c_j containing word w_i
- t_j: number of documents in D of class c_j
- $1, 2$: parameters for Laplace smoothing

For the Multinomial model, we also use maximum-likelihood estimation. Additionally, we apply **Laplace smoothing to avoid zero probabilities**.

$$P(\text{word}_i|\text{class}, D) = \frac{\text{tf}_{i,j} + \alpha}{\sum_{\text{doc}\in D} \text{tf}_j + \alpha \cdot |V|}$$

- $tf_{i,j}$: term frequency (n.occurrences) of w_i in document *doc* of class c_j
- tf_j: all-term frequency of document *doc* of class c_j
- α: smoothing parameter ($\alpha = 1$ for Laplace smoothing)

For text classification we can estimate the probabilities:

```
LEARN_NAIVE_BAYES_TEXT_BE($D, C$)
    $V \leftarrow$ all distinct words in $D$
    for each target value $c_j \in C$ do
        $docs_j \leftarrow$ subset of $D$ for which the target value is $c_j$
        $t_j \leftarrow |docs_j|$: total number of documents in $c_j$
        $\hat{P}(c_j) \leftarrow \frac{t_j}{|D|}$
        for each word $w_i$ in $V$ do
            $t_{i,j} \leftarrow$ number of documents in $c_j$ containing word $w_i$
            $\hat{P}(w_i|c_j) \leftarrow \frac{t_{i,j}+1}{t_j+2}$
```

Figure 5.1. Using Bernoulli distribution

```
LEARN_NAIVE_BAYES_TEXT_MU($D, C$)
    $V \leftarrow$ all distinct words in $D$
    for each target value $c_j \in C$ do
        $docs_j \leftarrow$ subset of $D$ for which the target value is $c_j$
        $t_j \leftarrow |docs_j|$: total number of documents in $c_j$
        $\hat{P}(c_j) \leftarrow \frac{t_j}{|D|}$
        $TF_j \leftarrow$ total number of words in $docs_j$ (counting duplicates)
        for each word $w_i$ in $V$ do
            $TF_{i,j} \leftarrow$ total number of times word $w_i$ occurs in $docs_j$
            $\hat{P}(w_i|c_j) \leftarrow \frac{TF_{i,j}+1}{TF_j+|V|}$
```

Figure 5.2. Using Multinomial distribution

5.4 Bayesian Learning: Summary

The foundation of Bayesian learning is that we can express a machine learning problem as a probabilistic one. $v^* = \mathbf{argmax}_{v \in V} P(v|x', D)$, **best classification** is: **class** with higher probability, with a x' new instance.

- **ML** , Maximum Likelihood hypothesis $h_{\text{ML}} = \mathbf{argmax}_{h \in H} P(D|h)$
  - Maximizes likelihood, seeks the most probable hypothesis given observed data.
- **MAP** , Maximum a Posteriori hypothesis $h_{\text{MAP}} = \mathbf{argmax}_{h \in H} P(h|D)$
  - Maximizes posterior probability, combining prior beliefs and likelihood of observed data, it extends ML hypothesis
  - **Brute force** : Calculate every $P(h|D)$ and take **maximizing** h.

Bayesian optimal classifier minimizes errors with perfect knowledge, since maximizing the dataset hypothesis doesn't ensure generalization. For $x \notin D$, class probability is the sum of instance-class and hypothesis weights.

$$P(v_j|x, D) = \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D)$$

Naive Bayes estimates class probability using conditional independence. It calculates the most probable class without explicit hypotheses, employing Bayes' rule:

$$v_{\text{NB}} = \arg\max_{v_j \in V} P(v_j|D) \prod_i P(a_i|v_j, D)$$

This assigns a class by evaluating conditional probabilities of attributes, multiplying with prior class probabilities, and taking the **argmax** for the final decision. This could be possible due to the **Naive Bayes assumption**

$$P(a_1, a_2, \ldots, a_n|v_j, D) = \prod_i P(a_i|v_j, D)$$

Naive Bayes for Text Classification

Documents are assigned to predefined classes like spam or not spam based on learned patterns. Naive Bayes **assumes word independence within a document class**, represented as the product of individual word-class probabilities. The "**bag of words**" method ignores word order, emphasizing presence or frequency. Two common feature vector representations are:

1. **Boolean Feature Vector**: Binary values indicate word presence (1) or absence (0), modeled by a Multivariate Bernoulli distribution.
2. **Ordinal Feature Vector** Elements represent word frequency, following a Multinomial distribution.

For the Multivariate Bernoulli model, probabilities are estimated using maximum-likelihood, with Laplace smoothing for nonzero probabilities. For the Multinomial model, Laplace smoothing is applied to handle zero probabilities.

Chapter 6

Probabilistic models for classification

In a classification problem, we have two different approaches

- **Generative**: Estimate $P(x|C_i)$ (class-conditional densities) and then compute $P(C_i|x)$ (posterior probability) with Bayes
- **Discriminative**: Estimate $P(C_i|x)$ (posterior probability) directly

6.1 Generative Models

6.1.1 Two Classes Case

Posterior probability (assuming that we have two classes):

$$P(C_1|x) = \frac{P(x|C_1)P(C_1)}{P(x)} = \frac{P(x|C_1)P(C_1)}{P(x|C_1)P(C_1) + P(x|C_2)P(C_2)} = \frac{1}{1+\exp(-a)} = \sigma(a)$$

Where we choose $a = \ln\left(\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)}\right)$

So basically we can model the computation of posterior probability as a **sigmoid function**, this sigmoid function takes as its argument the logarithm of the ratio of the probabilities of the two classes. The sigmoid function maps this log-odds ratio to a probability value between 0 and 1, allowing us to estimate the probability of an input belonging to one of the two classes.

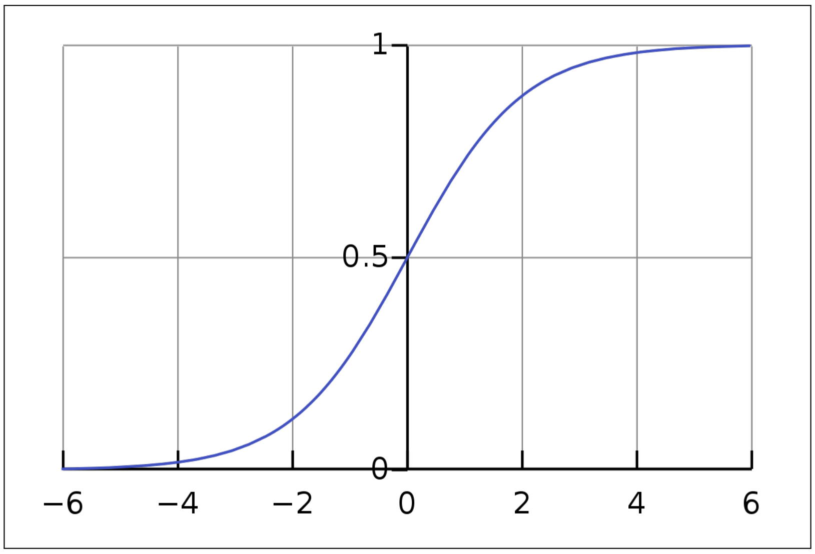


Figure 6.1. The sigmoid function graph

Now, to compute the **conditional probability** $P(x|C_i)$, we start from the assumption that the samples are generated following the Gaussian distribution.

$$P(x|C_i) = \mathcal{N}(x; \mu_i, \Sigma)$$

Where

- μ_i is the average of the Gaussian distribution for class C_i.
- Σ represents the covariance matrix of the Gaussian distribution for class C_i
  - In this model, this is the **same** for every class

We can substitute this in our parameter a definition previously given.

$$a = \ln\left(\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)}\right) = \ln\left(\frac{\mathcal{N}(x; \mu_1, \Sigma)P(C_1)}{\mathcal{N}(x; \mu_2, \Sigma)P(C_2)}\right)$$

Gaussian Distribution

Univariate Gaussian Distribution

For a single variable x with mean μ and variance σ^2:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Here,

- x: the variable.
- μ: the mean.
- σ^2: the variance.

Multivariate Gaussian Distribution Recalls

For a vector variable $\mathbf{x}$ with mean vector $\boldsymbol{\mu}$ and covariance matrix Σ:

$$P(\mathbf{x}) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

Here,

- $\mathbf{x}$: the vector variable.
- $\boldsymbol{\mu}$: the mean vector.
- Σ: the covariance matrix.
- D: the dimensionality of the vector.
- $|\Sigma|$: the determinant of the covariance matrix.

We want to riealaborate that in a **linear combination of terms** in x, to do so, we need to define some terms:

- $w = \Sigma^{-1}(\mu_1 - \mu_2)$
- $w_0 = -\frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln(P(C_1)P(C_2))$
  - *It essentially combines information about the means and covariances of the two classes and the class priors to determine which class a given data point is more likely to belong to*

So in the end, substituting the **Gaussian Formula** and then the parameters defined, we got

$$a = \ln\left(\frac{\mathcal{N}(x;\mu_1,\Sigma)P(C_1)}{\mathcal{N}(x;\mu_2,\Sigma)P(C_2)}\right) = w^Tx + w_0$$

$$P(C_1|x) = \sigma(w^Tx + w_0)$$

If we can **compute** w^T and w_0, we can essentially know the distribution of the classes over the samples,

6.1.2 Maximum Likelihood Solution (for 2 Classes)

Assumptions

Assuming:

$$P(C_1) = \pi$$
$$P(x|C_i) = \mathcal{N}(x; \mu_i, \Sigma)$$

Given dataset $D = \{(x_n, t_n)\}_{n=1}^{N}$, where

- $t_n = 1$ if x_n belongs to class C_1
- $t_n = 0$ if x_n belongs to class C_2
- N_1 be the number of samples in D belonging to C_1
- N_2 be the number of samples in D belonging to C_2

Likelihood Function (The Probability Distribution Function)

The likelihood function is given by:

$$P(t|\pi, \mu_1, \mu_2, \Sigma, D) = \prod_{n=1}^{N} \left[\pi\mathcal{N}(x_n; \mu_1, \Sigma)^{t_n} \cdot (1-\pi)\mathcal{N}(x_n; \mu_2, \Sigma)^{1-t_n}\right]$$

Likelihood Function

The likelihood function, denoted as $P(t|\pi, \mu_1, \mu_2, \Sigma, D)$, encapsulates the probability of observing the dataset D given specific parameters $\pi, \mu_1, \mu_2, \Sigma$ in the context of a two-class classification problem.
For each data point n in the dataset D, the likelihood function involves:

- $\pi\mathcal{N}(x_n; \mu_1, \Sigma)^{t_n}$: *Likelihood that x_n belongs to class C_1 if $t_n = 1$.*
- $(1-\pi)\mathcal{N}(x_n; \mu_2, \Sigma)^{1-t_n}$: *Likelihood that x_n belongs to class C_2 if $t_n = 0$.*

The overall likelihood function is a product of these individual likelihoods for all data points in the dataset D. The assumption is that data points are independent given the class labels. Formally, the likelihood function is expressed as:

$$P(t|\pi, \mu_1, \mu_2, \Sigma, D) = \prod_{n=1}^{N} \left[\pi\mathcal{N}(x_n; \mu_1, \Sigma)^{t_n} \cdot (1-\pi)\mathcal{N}(x_n; \mu_2, \Sigma)^{1-t_n}\right]$$

Here, t_n is the class label for data point n, taking the value 1 if the point belongs to class C_1 and 0 if it belongs to class C_2. The likelihood function combines the class priors π with the conditional probabilities modeled by multivariate Gaussian distributions for each class. The product reflects the assumption of independence among data points given their class labels.

Determine Parameters

To find the parameters $\pi, \mu_1, \mu_2, \Sigma$, you can maximize the log likelihood function:

$$\pi = \frac{N_1}{N}, \qquad \mu_1 = \frac{1}{N_1}\sum_{n=1}^{N} t_n x_n, \qquad \mu_2 = \frac{1}{N_2}\sum_{n=1}^{N}(1-t_n)x_n$$

And the covariance matrix Σ can be calculated as follows:

- $S_i = \frac{1}{N_i}\sum_{n\in C_i}(x_n-\mu_i)(x_n-\mu_i)^T \quad \text{for } i = 1,2$
- $\Sigma = \frac{N_1}{N}S_1 + \frac{N_2}{N}S_2$

Where S_i is the **scatter matrix** for class C_i. It quantifies the spread or dispersion of data points within the class, relative to the class mean μ_i. It sums over all the data points from the dataset that are part of class C_i. x_n represents an individual data point, in the context of the summation, it takes on the values of each data point within class.

6.1.3 K-class case

The K-class case is just an extension of the previosly discussed **Two Class** case. The difference is that the **posterior probability** is not equal to the **sigmoid function**

$$P(C_k|x) = \frac{P(x|C_k)P(C_k)}{\sum_j P(x|C_j)P(C_j)} = \frac{\exp a_k}{\sum_j \exp a_j}$$

Where $a_k = \ln(P(x|C_k)P(C_k)) = w^T x + w_0$

Compact Notation

$$w^T x + w_0 = \begin{pmatrix} w_0 & w \end{pmatrix}\begin{pmatrix} 1 \\ x \end{pmatrix}$$

$$\tilde{w} = \begin{pmatrix} w_0 \\ w \end{pmatrix}, \quad \tilde{x} = \begin{pmatrix} 1 \\ x \end{pmatrix}$$

$$a_k = w^T x + w_0 = \tilde{w}^T \tilde{x}$$

6.2 Discriminative Models

The discriminative approach in probabilistic models prioritizes classifying data accurately but doesn't allow for generating new data, they may not provide a full probabilistic description of the data, and as a result, they may not be well-suited for generating new data points. Generative models, on the other hand, aim to model the entire data distribution, allowing for the generation of new samples.

6.2.1 Two-Class Logistic Regression

Even if the name "**Regression**" is often associated to models in the continuous space, here we will refer to classification problems. The **Maximum likelihood function** will be

$$P(t|\tilde{w}) = \prod_{n=1}^{N} y_n^{t_n} (1 - y_n)^{1-t_n}$$

Where

- t_n is the actual value (class) in the **dataset** corresponding to x_n,
- y_n is the posterior prediction of the current model $\tilde{w}$ for x_n.
- $y_n = p(C_1|\tilde{x}_n) = \sigma(\tilde{w}^T \tilde{x}_n)$

For this model, the **cross-entropy error** function is equal to:

$$E(\tilde{w}) \equiv -\ln p(t|\tilde{w}) = -\sum_{n=1}^{N} [t_n \ln y_n + (1 - t_n) \ln(1 - y_n)]$$

Our goal is to minimize this function

$$\tilde{w}^* = \arg\min_{\tilde{w}} E(\tilde{w})$$

This is basically the solution of a **nonlinear** function. That can always be written as the research of the roots of a nonlinear function, so in this case it's possible to apply the **Newton-Rhapson** and other methods.

Extend the method to predict multiple classes using 1-out-of-k encoding for output representation, then solve the optimization problem. It's versatile and works with transformed input spaces.

6.3 Probabilistic models for classification: Summary

Two different approaches for a classification problem:

- **Generative**: Estimate $P(x|C_i)$ and **compute** $P(C_i|x)$
- **Discriminative**: Estimate $P(C_i|x)$ directly.

Generative Models

Assuming a **two classes** case, $P(C_i|x) = \sigma(a)$ where $a = \ln\left(\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)}\right)$
The computation of posterior probability is modeled with a **sigmoid function.** It maps this log-odds ratio to a probability, estimating class membership between 0 and 1. To compute the **conditional probability** we need the **assumption of a Gaussian Distribution** of the samples $P(x|C_i) = \mathcal{N}(x; \mu_i, \Sigma)$
We want to riealaborate that in a **linear combination of terms** in x, to do so, we need to define some terms:

- $w = \sum^{-1}(\mu_1 - \mu_2)$
- $w_0 = -\frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln(P(C_1)P(C_2))$

Substituting the **Gaussian Formula** and then the parameters defined:

- $a = \ln\left(\frac{\mathcal{N}(x;\mu_1,\Sigma)P(C_1)}{\mathcal{N}(x;\mu_2,\Sigma)P(C_2)}\right) = w^Tx + w_0$
- $P(C_1|x) = \sigma(w^Tx + w_0)$

The **Maximum likelihood solution**, assuming **Gaussian distirbution** we have the likelihood function in the form of

$$P(t|\pi, \mu_1, \mu_2, \Sigma, D) = \prod_{n=1}^{N}\left[\pi\mathcal{N}(x_n;\mu_1,\Sigma)^{t_n} \cdot (1-\pi)\mathcal{N}(x_n;\mu_2,\Sigma)^{1-t_n}\right]$$

The goal is finding the parameters $\pi, \mu_1, \mu_2, \Sigma$:

$$\pi = \frac{N_1}{N}, \qquad \mu_1 = \frac{1}{N_1}\sum_{n=1}^{N} t_n x_n, \qquad \mu_2 = \frac{1}{N_2}\sum_{n=1}^{N}(1-t_n)x_n$$

$$\Sigma = \tfrac{N_1}{N}S_1 + \tfrac{N_2}{N}S_2 \text{ where}$$
$$S_i = \tfrac{1}{N_i}\textstyle\sum_{n\in C_i}(x_n - \mu_i)(x_n - \mu_i)^T \quad \text{for } i = 1,2 \text{ (\textbf{scatter matrix})}$$

Discriminative Models

Discriminative models prioritize accurate classification but lack data generation capability. They may not offer a complete probabilistic description of data.
For **Two-Class Logistic Regression** :
Maximum likelihood function: $P(t|\tilde{w}) = \prod_{n=1}^{N} y_n^{t_n}(1-y_n)^{1-t_n}$ Where t_n is the actual class in the dataset for x_n, and y_n is the posterior prediction of the model $\tilde{w}$ for x_n ($y_n = p(C_1|\tilde{x}_n) = \sigma(\tilde{w}^T\tilde{x}_n)$).
Cross-entropy error function:

$$E(\tilde{w}) = -\ln p(t|\tilde{w}) = -\textstyle\sum_{n=1}^{N}\left[t_n \ln y_n + (1-t_n)\ln(1-y_n)\right]$$

Goal: Minimize $E(\tilde{w})$ for $\tilde{w}^* = \arg\min_{\tilde{w}} E(\tilde{w})$, a solution to a nonlinear function. Methods like Newton-Raphson can be applied.

Chapter 7

Linear Models for Classification

The main asssumption when working with linear models is that the samples in the dataset have the property of being linearly separable.

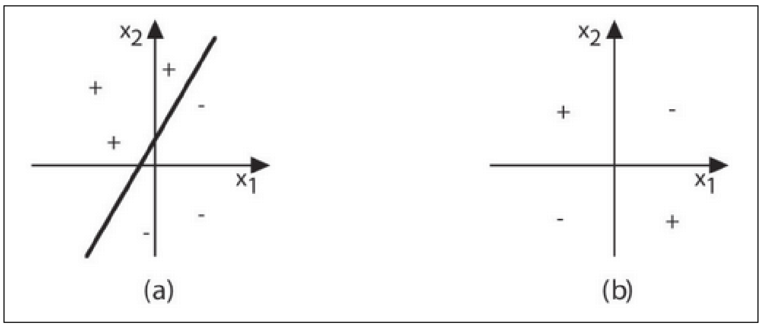


Figure 7.1. Separable and Non-Separable Dataset Example

We say that a dataset is linerarly separable if exists a linear function (a **Hyperplane**) that separates the space in two or more region, where each region belongs to a certain class.

Compact Notation

Two classes

$$y(x) = w^T x + w_0 = \tilde{w}^T \tilde{x}, \text{ with:}$$
$$\tilde{w} = \begin{pmatrix} w_0 \\ w \end{pmatrix}, \ \tilde{x} = \begin{pmatrix} 1 \\ x \end{pmatrix}$$

w is the **weights** vector, in the 2D case we have just. the **slope**.

K classes
One function per class

$$y(x) = \begin{pmatrix} y_1(x) \\ \vdots \\ y_K(x) \end{pmatrix} = \begin{pmatrix} w_1^T x + w_{10} \\ \vdots \\ w_K^T x + w_{K0} \end{pmatrix} = \begin{pmatrix} \tilde{w}_1^T \\ \vdots \\ \tilde{w}_K^T \end{pmatrix} \tilde{x} = \tilde{W}\tilde{x}, \text{ with: } \tilde{W}^T = \begin{pmatrix} \tilde{w}_1^T \\ \vdots \\ \tilde{w}_K^T \end{pmatrix}$$

Binary classifiers *(e.g., one-vs-rest) for* ***multi-class problems*** *often lead to* ***ambiguous regions without class assignments****, causing decision conflicts. This approach lacks a clear classification for data points in unassigned areas, making it less effective. Similarly, one-vs-one classifiers face similar issues, making them less desirable for multi-class classification.*

For Example inn One-vs-Rest classification for dogs, cats, and birds, three classifiers—Dog vs. Non-Dog, Cat vs. Non-Cat, Bird vs. Non-Bird—may leave a photo unclassified in ambiguous regions.

The correct approach involves defining the classifier as a combination of k linear models, which is **ideal for multi-class classification**. Each sample is classified based on the class with the **highest prediction value** , ensuring a proper division of the space into k classes. This approach leads to effective k-class discriminants, as demonstrated by various model solutions.

7.1 Approaches to learn linear discriminant

7.1.1 Least Squares

Given dataset $D = \{(x_n, t_n)\}_{n=1}^{N}$, with a 1-out-of-k encoding. The error function for this model is defined as the **sum of the squared errors**, where the error is represented as the distance between the **prediction** $\tilde{X}\tilde{W}$ and the truth T

$$E(\tilde{W}) = \frac{1}{2}\text{Tr}\left((\tilde{X}\tilde{W} - T)^T(\tilde{X}\tilde{W} - T)\right)$$

Where

- Tr stands for the trace of the matrix, sum of the elements on its main diagonal (the sum of the squared errors).
- 1/2 is there just for the purpose of being simplified when computing the derivative of the squared error. . When you differentiate the error function to find its minimum, the term cancels out and simplifies the calculations. This factor is often included for mathematical convenience and doesn't affect the optimization process's result, as the minimum point remains the same.

We have to **minimize** the error function we just defined, if we compute the derivative to obtain the minimizing weights we get the **closed-form solution** :

$$\tilde{W} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T T$$

Once the solution is computed, we will use the obtained weights inside our model to make predictions:

$$y(X) = \tilde{W}^T\tilde{X} = T^T(\tilde{X}^{\dagger})^T\tilde{X}$$

Where $X^\dagger$ represents the **Hermitian transpose** of X, we will have a full vector of these predictions, the final predicted class will be $C_k = \textbf{argmax}_{i=1,\dots,k} y_i(x)$

The fundamental problem with the least squares method is that the linear separator depends only on the distance error, meaning that outliers will tend to move the line towards them, making the entire model less performant:

7.1.2 Perceptron

A perceptron is a type of artificial neural network model and is one of the simplest forms of a **neural network**, a perceptron unit is made by pipeline of two operations.

1. Computing **weighted linear combination** of the inputs ($\Sigma = \sum_i w_i x_i$)
2. **Applying sign function**, $y(x) = sign(W^T X)$

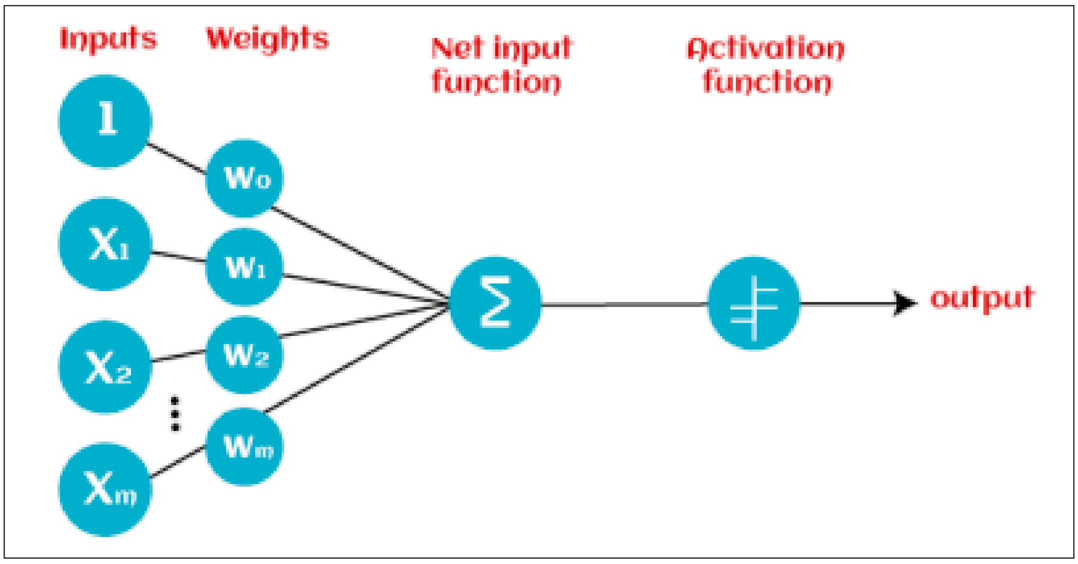


Figure 7.2. Perceptron scheme

We aim, with dataset $D = \{(x_n, t_n) \,|\, n = 1, 2, \dots, N\}$ and the error function that we define as sum of the squared errors, to find the w_i which minimize the function

$$E(W) = \frac{1}{2} \sum_{n=1}^{N} (t_n - w^T x_n)$$

In order to find the minimum of the error function we start by computing its gradient

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{n=1}^{N} (t_n - w_n^T x_n)^2 = \frac{1}{2} \sum_{n=1}^{N} \frac{\partial}{\partial w_i} (t_n - w_n^T x_n)^2$$

$$\sum_{n=1}^{N} (t_n - w_n^T x_n)(-x_{i,n})$$

NOTE that in this case we have **a weight for every input** , the negative gradient will show us the direction of maximum decrease, that we will follow using an iterative approach in order to reach the minimum

here are the **update rule** for the **perceptron**

$$w_i \leftarrow w_i + \Delta w_i, \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Where η is called **learning rate** , it represents how much we move towards the negative direction of the gradient. Including the **sign function** (and that means, including a **threshold**)

This formulation makes the computation of the model really robust to outliers, since even extreme values will be mapped to 1 or -1. The error will be 0 when the line will perfectly separate the data. The iterations will proceed for every **missclassified data point** in the dataset

$$\Delta w_i = \eta \sum_{n=1}^{N} (t_n - \mathbf{sign}(w_n^T x_n))x_{i,n}$$

Implementations

When feeding the data to the training phase, we have three possible approaches:

- Batch mode: Compute Δw_i with the whole dataset
- Mini-Batch mode: Choose a small subset $S \subset D$ at a time to compute Δw_i
- Incremental mode: Choose one sample $(x, t) \in D$ at a time to compute Δw_i

The most effective training approach for linear models in classification, given certain conditions, it's usually the mini-batch mode. It strikes a balance between computational efficiency and adaptability to large datasets, making it a practical choice. In this mode, weight updates are calculated by considering a small subset of data at a time, denoted as a mini-batch.

The conditions necessary for algorithm convergence include the requirement that the data must be **linearly separable** , and the learning rate needs to be carefully chosen to ensure convergence.

In the mini-batch mode, weight updates are determined by considering the contributions from samples within the mini-batch. The update equation takes into account the difference between the true target values and the model's predictions. The learning rate should be appropriately chosen, as a **large learning rate** can lead to instability, while **a small one** may lead to **overfitting** .

The training process terminates under some conditions:

- The error reaches **zero** (representing an optimal solution, though not always attainable)
- The change in the loss function is smaller than a **predefined threshold**
- The predefined number of iterations for the training are done.

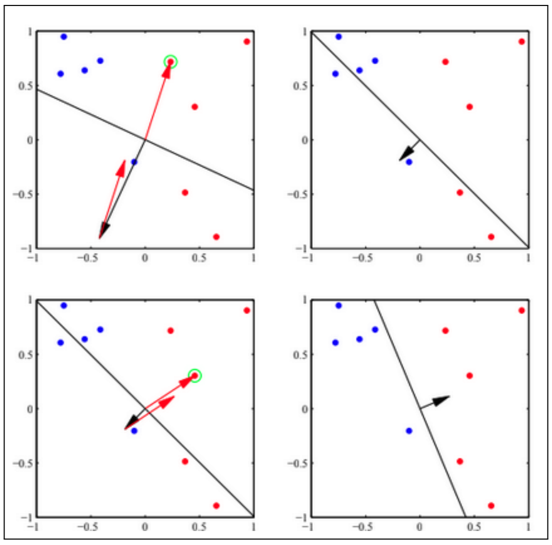


Figure 7.3. Visualization of a learning process (binary classification)

7.1.3 Fisher's linear discriminant

The core idea about Fisher's linear discriminant is to place the linear separator in the middle of the line that connects the centers of mass (means) of the distribution

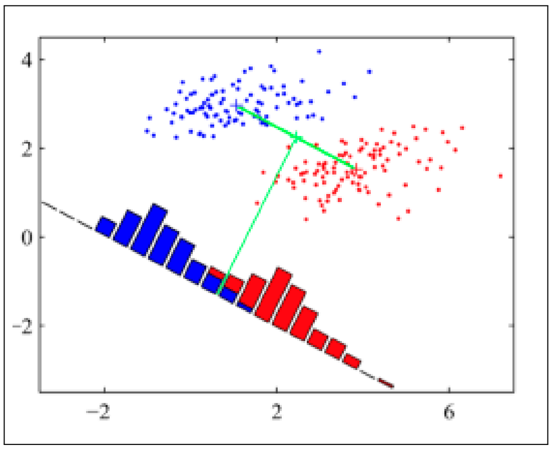


Figure 7.4. Fisher Linear Discriminant

Let's consider a dataset with N_1 elements belonging to class C_1 and N_2 elements of class C_2.

The **separation** of the two classes will be expressed as $J(w) = \frac{w^T S_B w}{w^T S_W w}$

Where

- $S_B = (m_2 - m_1)(m_2 - m_1)^T$ is the **between-class scatter**
  - measures how much classes are far apart on average
- $S_W = \sum_{n \in C_1}(x_n - m_1)(x_n - m_1)^T + \sum_{n \in C_2}(x_n - m_2)(x_n - m_2)^T$ is the **whithin-**

class scatter

- m_1 and m_2 in those formulas represent the **mean** of the two classes.

Our goal is to choose w that maximizes $J(w)$, by solving $\frac{d}{dw}J(w) = 0$.

Rotation Matrix

The current solution has the major flaw of not taking into account the covariance of the two distributions. A straight forward solution is to introduce an additional parameters to the optimization problem **in the form of a rotation matrix**. The result will be a rotation of the linear separator, that will provide a better split for the two regions.

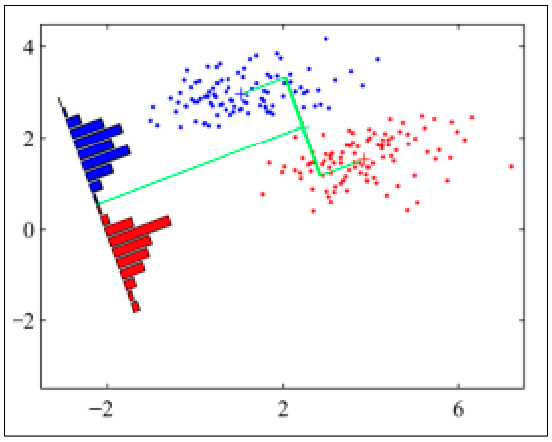


Figure 7.5. Rotation Applied

7.1.4 Support Vector Machines (SVMs)

Support Vector Machines were born with the idea of addressing the problem of finding the best division in space in order to provide better accuracy. The core concept for this model lies into the definition of margins. A margin is the distance between the closest point of the dataset and the linear separator, maximum margin providing for better accuracy.

Assume D is linearly separable, i.e., there exist w and w_0 such that for all $n = 1, \dots, N$.

$$y(x_n) > 0 \text{ if } t_n = +1, \qquad y(x_n) < 0 \text{ if } t_n = -1,$$

We define the margin of a point as $\frac{|y(x_l)|}{\|w\|}$, namely the distance between a certain point and the separator, using the concept of margin, we will define our optimization problem in the terms of maximizing the margin from the closest point to the line, using the property $|y(x)| = t_n y(x_n)$ and substituing our linear form, the **margin of the closest** point is:

$$\min_{n=1,\dots,N} \frac{|y(x_n)|}{\|w\|} = \dots = \frac{1}{\|w\|} \min_{n=1,\dots,N} \left[t_n(\bar{w}^T x_n + \bar{w}_0) \right]$$

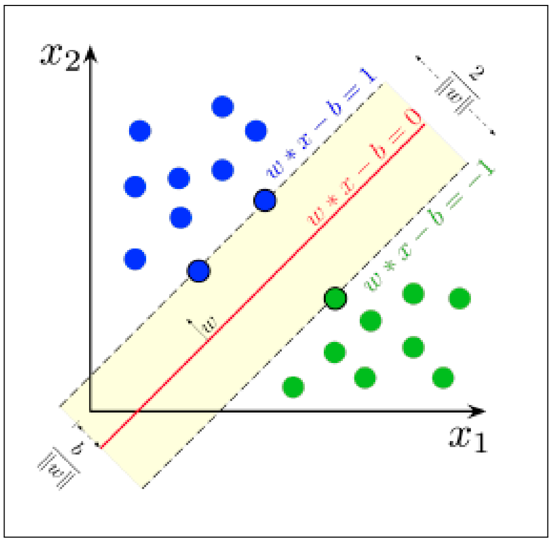


Figure 7.6. Visualization of the Margin

$$(w^*, w_0) = \arg\max \frac{1}{\|w\|} \min \left[t_n \left(w^T x_n + w_0\right)\right]_{n=1,\dots,N}$$

A fundamental step before computing the solution is the **normalization of the dataset**, **rescaling** all the points does not affect the solution. This means to scale all the points such that the minimum margin for the closest point is greater or equal than 1. Once normalized, we will have that we will have at least two samples (one for C_1, the other for C_2) having a margin of 1.

So basically we have two traslated:

- An Hyperplane touching Support Vector with margin 1: $W^{*T} x_k^+ + w_0 = +1$
- An Hyperplane touching Support Vector with margin -1: $W^{*T} x_k^- + w_0 = -1$
- The Separation Hyperplane $W^{*T} x_k^- + w_0 = 0$

Since the distance between a point and a hyperplane is $d = \frac{|w^t x + b|}{||w||}$ we know that this is equal to $\frac{1}{||w||}$ for these support vector, so the total width is $\frac{2}{||w||}$

In the canonical representation of the problem the maximum margin hyperplane can be found by solving the optimization problem, with respect just to these two points. The optimization problem is formulated as:

$$(w^*, w_0^*) = \arg\max \frac{1}{\|w\|} = \arg\min \frac{1}{2}\|w\|^2$$

The solution to this problem can be computed using the Lagrangian Multipliers method:

$$a^* = \sum_{n=1}^{N} a_n^* t_n x_n$$

The method of Lagrange multipliers is an approach to find the maximum or minimum points of a function subject to constraints. Specifically, in the context of SVMs, it is used to solve the optimization problem by **considering the constraints that define the margins.** The solution to this optimization problem leads to the determination of the optimal parameters of the hyperplane (w^*, w_0^*), which maximize the margin between the classes.

The terms a_n^* are named Lagrange multipliers, and they will be our unknown term.

$$\begin{gathered}\tilde{L}(a) = \textstyle\sum_{n=1}^{N}(a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m t_n t_m x_n^T x_m) \\ \text{subject to} \quad a_n \geq 0 \quad \forall n = 1, \dots, N \\ \textstyle\sum_{n=1}^{N} a_n t_n = 0\end{gathered}$$

Samples x_n for which $a_n^* = 0$ do not contribute to the solution.

Karush-Kuhn-Tucker (KKT) Condition:

In mathematics, the Karush-Kuhn-Tucker conditions (also known as Kuhn-Tucker conditions or KKT conditions) are necessary conditions for solving a nonlinear programming problem when the constraints satisfy one of the regularity conditions called constraint qualification conditions. For each $x_n \in D$, either:

- $a_n^* = 0$
- $t_n \cdot y(x_n) = 1$

Thus, if $t_n \cdot y(x_n) > 1$, it implies $a_n^* = 0$.

We define **Support Vectors** x_k, vectors such that $t_k \cdot y(x_k) = 1$ and $a_k^* > 0$.

SV being the sets of Support Vectors, the only terms that ***will contribute to the solution****. formally defined as*

The **hyperplane equation** is defined as

$$y(x) = \sum_{x_j \in SV} a_j^* t_j x^T x_j + w_0^* = 0$$

More on this equation

To account for all support vectors and their contributions to the decision function, we can represent $w^T x$ as a linear combination of dot products:

$$w^T x = \sum_j a_j^* t_j x_j^T x + b$$

This representation essentially states that $w^T x$ can be reconstructed as a weighted sum of dot products between the support vectors (x_j) and the input vector (x), weighted by the Lagrange multipliers (a_j^*) and scaled by their corresponding target labels (t_j).

We need to compute the **bias term** w_0^*, in order to do so, we pick any support vector and solve the equation:

$$w_0^* = t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j$$

Or maybe with respect to the **mean of all support vectors**

$$w_0^* = \frac{1}{|SV|} \sum_{x_k \in SV} \left(t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j \right)$$

Once we find the maximum margin hyperplane, determined by a_k^*, w_0^* every new instance will be classified by applying the model:

$$y(x') = \text{sign} \left(\sum_{x_k \in SV} a_k^* t_k x'^T x_k + w_0^* \right)$$

SVMs were previously acknowledged for their **robustness to noise** caused by outliers, as points lying outside the margin have no impact on the optimization problem's solution. However, there is another type of noise that can pose challenges for SVMs—**points that are extremely close** to the separator relative to the distribution of their class. A evenly distant separator would **maximize the probability of accurately classifying new instances** by dividing the space more evenly, and to achieve that, one approach sometimes is to consider **relaxing the original constraints** of the optimization problem by allowing for some exceptions.

Let us introduce **slack variables** $\xi_n \geq 0$, $n = 1, \ldots, N$:

- $\xi_n = 0$ if the point is on or inside the correct margin boundary.
- $0 < \xi_n \leq 1$ if the point is inside the margin but on the correct side.
- $\xi_n > 1$ if the point is on the wrong side of the boundary.

When $\xi_n = 1$, the sample lies on the decision boundary ($y(x_n) = 0$).

When $\xi_n > 1$, the sample will be mis-classified.

The soft margin constraint is given by:

$$t_n y(x_n) \geq 1 - \xi_n, \quad n = 1, \dots, N.$$

The optimization problem with soft margin constraints can be written as:

$$w^*, w_0^* = \arg\min \frac{1}{2}||w||^2 + C\sum_{n=1}^{N} \xi_n.$$

C is a constant (inverse of a regularization coefficient)

Finally, the solution can still be computed using the methods explained before (the lagrangian solution and the KKT condition are still valid).

7.1.5 Not Linearly separable dataset

We already introduced the fact that linear classfication problems works also if the input space is transformed. This means that even if the dataset might not be linearly separable, we can still try to apply some **basis transformation** in order to make it work. That can increase or decrease dimensions.

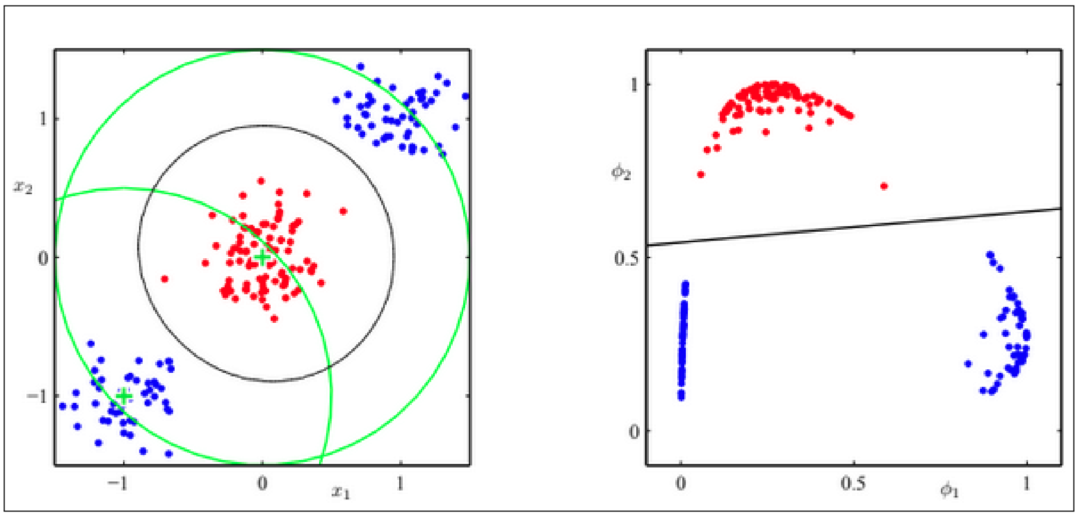


Figure 7.7. Coordinate change to make the dataset linearly separable

- **Original Dataset:** $D = \{(x_n, t_n)\}_{n=1}^{N}$ in space $\mathbb{R}^d$.
- **Transformation Function:** $\phi(x) = \begin{bmatrix} \phi_0(x) \\ \vdots \\ \phi_M(x) \end{bmatrix}$, where $M \neq d$
- **Transformed Dataset:** $D_\phi = \{(\phi_n, t_n)\}$, making it linearly separable.

7.2 Linear Models for Classification: Summary

A Dataset is **linearly separable** if exists a linear function **(Hyperplane)** that separates the space in two or more regions, where each region belongs to a certain class.
K classes, one classifying function per class

$$y(x) = \begin{pmatrix} y_1(x) \\ \vdots \\ y_K(x) \end{pmatrix} = \begin{pmatrix} w_1^T x + w_{10} \\ \vdots \\ w_K^T x + w_{K0} \end{pmatrix} = \begin{pmatrix} \tilde{w}_1^T \\ \vdots \\ \tilde{w}_K^T \end{pmatrix} \tilde{x} = \tilde{W}\tilde{x}, \text{ with: } \tilde{W}^T = \begin{pmatrix} \tilde{w}_1^T \\ \vdots \\ \tilde{w}_K^T \end{pmatrix}$$

Using **binary classifiers** like one-vs-rest or one-vs-one for multi-class problems **leads to ambiguous regions without class assignments**, the correct approach is using a combination of **k** linear models and each sample is classified based on the class with the **highest prediction value**.
Least Squares, minimize the **sum of squared errors** between the prediction $\tilde{X}\tilde{W}$ and the truth T, to obtain the minimizing weights we get the **closed form solution** $\tilde{W} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T T$ and then we make the predictions $y(X) = \tilde{W}^T\tilde{X} = T^T(\tilde{X}^\dagger)^T\tilde{X}$, this method is really sensitive to **outliers**.
Perceptron, one of the simplest form of Artificial Neural Networks, pipeline of two operations:

1. Computing **weighted linear combination** of the inputs ($\Sigma = \sum_i w_i x_i$)
2. **Applying sign function**, $y(x) = sign(W^T X)$

We aim to minimize $E(W) = \frac{1}{2}\sum_{n=1}^{N}(t_n - w^T x_n)$ over time, we compute the gradient $\sum_{n=1}^{N}(t_n - w_n^T x_n)(-x_{i,n})$ and then apply **Gradient Descent** in an iterative way.

$$w_i \leftarrow w_i + \Delta w_i, \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Learning rate η adjusts gradient descent step size. **Sign function** with threshold enhances **robustness to outliers**, targeting error-free separation. **Iterations based on misclassified points** .
Fisher's Linear Discriminant, Maximize $J(w) = \frac{w^T S_B w}{w^T S_W w}$, where $S_B = (m_2 - m_1)(m_2 - m_1)^T$, $S_W = \sum_{n \in C_1}(x_n - m_1)(x_n - m_1)^T + \sum_{n \in C_2}(x_n - m_2)(x_n - m_2)^T$, by solving $\frac{d}{dw}J(w) = 0$.
Support Vector Machines, maximize the **margin** from the closest points to the separator, after **rescaling** the dataset we can write
$(w^*, w_0^*) = \arg\min \frac{1}{2}\|w\|^2$, we can solve it by using the **lagrande multipliers** method with the **KTT** conditions. **bias** computed using a single SV or mean

$$w_0^* = t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j \qquad w_0^* = \frac{1}{|SV|}\sum_{x_k \in SV}\left(t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j\right)$$

nce we find the maximum margin hyperplane, we can apply the model

$$y(x') = \text{sign}\left(\sum_{x_k \in SV} a_k^* t_k x'^T x_k + w_0^*\right)$$

Chapter 8

Linear Regression

Linear models for regression aim to finding models able to predict **real values** . Learning a function $f : X \to Y$, with $X \subseteq \mathbb{R}^d$ and $Y = \mathbb{R}$ from the dataset $D = \{(x_n, t_n)\}_{n=1}^{N}$.

When considering this problem, transformation of the input space are allowed, since we will still keep the model linear with respect to the parameters. Therefore, we can defines a **set of nonlinear transformations** and use it in the general formulation of linear models

$$y(x; w) = \sum_{j=0}^{M} w_j \varphi_j(x) = w^T \varphi(x)$$

,

$$\text{with } w = \begin{bmatrix} w_0 \\ \vdots \\ w_M \end{bmatrix}, \ \varphi(x) = \begin{bmatrix} \varphi_0(x) \\ \vdots \\ \varphi_M(x) \end{bmatrix}, \text{ and } \varphi_0(x) = 1.$$

Usually, the more parameter we use, the more we will risk the phenomena of **overfitting** . In fact we will progressively reduce the error on training dataset, but we will have poor generalization for new predictions.

8.1 Linear Regression Algorithms

8.1.1 Maximum likelihood and least squares

We start by assuming that points of the training dataset come from the true function plus some noise: $t = y(x; w) + \epsilon$.

The logic is that we assume that there is a true function, and each point of the datase correspond to the points of that function plus some noise error generated by a certain distribution that we **assume to be Gaussian**

$$P(\epsilon|\beta) = \mathcal{N}(\epsilon|0, \beta^{-1})$$

With **precision** (**inverse variance** $\frac{1}{\sigma^2}$) β, so we have $P(t|x,w,\beta) = \mathcal{N}(t|y(x;w),\beta^{-1})$

We also assume **observations independent and identically distributed**. We seek the maximum of the **likelihood function**:

$$P(\{t_1,\dots,t_N\}|x_1,\dots,x_N,w,\beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|w^T\varphi(x_n),\beta^{-1})$$

Or we can use the **negative log-likelihood** As discussed in the previous chapters, that corresponds to **minimizing the squared error between the actual data and our model**.

$$\ln P(t_1,\dots,t_N|x_1,\dots,x_N,w,\beta) = -\frac{\beta}{2}\sum_{n=1}^{N}\left[t_n - w^T\phi(x_n)\right]^2 - \frac{N}{2}\ln(2\pi\beta^{-1})$$

We can **search the minimum** of this function but the term we care about is just $\sum_{n=1}^{N}\left[t_n - w^T\phi(x_n)\right]^2$, so we can reduce everything to

$$\operatorname{argmin} \sum_{n=1}^{N}\left[t_n - w^T\phi(x_n)\right]^2$$

Computing Solution: Closed-form approach

The closed-form solutions aims at solving the error by directly computing the solution for the parameters matrix. We start defining the error:

$$E_D(w) = \frac{1}{2}(t-\Phi w)^T(t-\Phi w)$$

$$\text{with } t = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \text{ and } \Phi = \begin{bmatrix} \varphi_0(x_1) & \varphi_1(x_1) & \dots & \varphi_M(x_1) \\ \varphi_0(x_2) & \varphi_1(x_2) & \dots & \varphi_M(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_0(x_N) & \varphi_1(x_N) & \dots & \varphi_M(x_N) \end{bmatrix}.$$

Optimality condition, we trying to **minimize** the error function:

$$\nabla E_D = 0 \iff \Phi^T\Phi w = \Phi^T t$$

. Hence: $W_{\text{ML}} = (\Phi^T\Phi)^{-1}\Phi^T$ t, where $(\Phi^T\Phi)^{-1}\Phi^T = \Phi^\dagger$ is the Moore-Penrose pseudo-inverse of Φ.

Computing Solution: Iterative approach

In an ideal scenario, the closed-form approach would provide the correct solution. However, the challenge with this method is that it attempts to solve the problem for the entire dataset, which can lead to significant computational time issues, especially when dealing with high-dimensional data.

To overcome these performance issues, the iterative approach takes a step-by-step approach, gradually finding the negative gradient direction and updating the weights at each iteration. This method is commonly known as "**stochastic gradient descent** ." It offers a more practical and efficient way to optimize solutions, particularly in situations involving large or high-dimensional datasets.

$$\hat{w} \leftarrow \hat{w} - \eta \nabla E_n$$

Just as we saw in the previous chapter, η is called **learning rate**.

$$\hat{w} \leftarrow \hat{w} + \eta[t_n - \hat{w}^T \varphi(x_n)]\varphi(x_n)$$

Regularization is a technique employed to combat **overfitting**. The primary goal is to find the model parameters that minimize the empirical risk and maintain good generalization performance. One common approach to regularization is to **modify the optimization problem by introducing aregularization term** , which is typically added to the empirical risk.

The modified optimization problem is often expressed as:

$$\arg\min_{w} \left(E_D(w) + \lambda E_W(w)\right)$$

Here, $\lambda > 0$ represents the regularization factor, which controls the **trade-off between fitting the data well and keeping the model parameters in check.** The term $E_D(w)$ stands for the empirical risk or the loss function, which measures the model's fit to the training data.

A common choice for the regularization term $E_W(w)$ is the **L2** regularization, which can be expressed as:

$$E_W(w) = \frac{1}{2} w^T w$$

Alternatively, other choices for the regularization term, such as the **L1** regularization, can be used, which can be expressed as:

$$E_W(w) = \sum_{j=0}^{M} |w_j|^q$$

The purpose of introducing this regularization term is to prevent overfitting by penalizing model coefficients that are too high, regardless of the samples in the dataset. In this way, regularization encourages a balance between fitting the training data and ensuring that the model parameters remain within reasonable bounds.

8.2 Linear Regression: Summary

Linear Regression models aim to be able to predict real values based on a Dataset. We are basically **approximating** the real function between the data with the **best linear approximation** possible.

Maximun Likelihood and least squares
We assume points of the dataset as derived from a **function** plus some **gaussian noise** , $t = y(x; w) + \epsilon$. We also assume **observations independent and identically distributed**, we aim to max the **maximum likelihood** (equivalent to minimize the **log-likelihood)**

$$\ln P(t_1, \dots, t_N | x_1, \dots, x_N, w, \beta) = -\frac{\beta}{2} \sum_{n=1}^{N} \left[t_n - w^T \phi(x_n)\right]^2 - \frac{N}{2} \ln(2\pi\beta^{-1})$$

$$\text{argmin} \sum_{n=1}^{N} \left[t_n - w^T \phi(x_n)\right]^2$$

Computing Solution: Closed Form approach
We directly compute the solution from the parameter matrix, $\nabla E_D = 0 \iff \Phi^T \Phi w = \Phi^T t$, we obtain $W_{\text{ML}} = (\Phi^T \Phi)^{-1} \Phi^T$
Computing Solution: Iterative approach
We apply the **stochastic gradient descent** , $\hat{w} \leftarrow \hat{w} - \eta \nabla E_n$ where η is called **learning rate**. $\hat{w} \leftarrow \hat{w} + \eta[t_n - \hat{w}^T \varphi(x_n)]\varphi(x_n)$
Regularization
This is a method to combat **overfitting** by penalizing high coefficients

$$\arg\min_w (E_D(w) + \lambda E_W(w))$$

Here, $\lambda > 0$ represents the regularization factor, which controls the **trade-off between fitting the data well and keeping the model parameters in check.** The term $E_D(w)$ stands for the empirical risk or the loss function, which measures the model's fit to the training data.
A common choice for the regularization term $E_W(w)$ is the **L2** regularization, which can be expressed as:

$$E_W(w) = \frac{1}{2} w^T w$$

Alternatively, other choices for the regularization term, such as the **L1** regularization, can be used, which can be expressed as:

$$E_W(w) = \sum_{j=0}^{M} |w_j|^q$$

Chapter 9

Kernel Methods

In previous chapters, we focused on input spaces with known and fixed dimensions, but in real-world scenarios, this is not always the case. In practice, we frequently encounter input spaces of variable lengths and, in some cases, even infinite dimensions, like those associated with strings, image features, and more.

This is where **kernel functions** become essential. Essentially, kernel functions are mathematical functions that, when given two inputs, yield a real value representing their similarity. Defined as:

$$k : X \times X \to \mathbb{R}$$

$$k(x, x') \text{ similarity of input samples } x \text{ and } x'$$

Linear kernel: $k(x, x') = x^T x'$

9.1 Linear Models with and without Kernel

Without Kernel:

- **Linear model**

$$y(x; \boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n x_n^T x$$

- **Solution**

$$\boldsymbol{\alpha} = (K + \lambda I_N)^{-1} \mathbf{t}$$

- **Gram Matrix**

$$K = \begin{bmatrix} x_1^T x_1 & \dots & x_1^T x_N \\ \vdots & \ddots & \vdots \\ x_N^T x_1 & \dots & x_N^T x_N \end{bmatrix}$$

With Kernel:

- **Linear model with any kernel k**

$$y(x; \boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n k(x_n, x)$$

- **Solution**

$$\boldsymbol{\alpha} = (K + \lambda I_N)^{-1} \mathbf{t}$$

- **Gram Matrix**

$$K = \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{bmatrix}$$

9.2 Kernel Trick

Kernel Trick or Kernel Substitution If an algorithm involves input vectors in the form of inner products $x^T x'$, you can replace the inner product with a kernel function $k(x, x')$.

- Applicable to any input vector x, even if it has an infinite size.
- No need to **explicitly know the feature mapping** $\phi(x)$.
- Allows for the direct extension of many well-known algorithms.

In this context, ϕ represents a feature mapping function that transforms input data into a potentially higher-dimensional space. The kernel trick allows for similarity calculations between data points in this higher-dimensional space without the need to explicitly compute $\phi(x)$. This simplifies computations, particularly when $\phi(x)$ is complex or unknown, making it a valuable tool for handling non-linear patterns in machine learning and data analysis.

The more the two inputs are similiar, the more the output value of the kernel will be close to zero . The kernel trick is pivotal for handling non-linear patterns, dimension reduction, and diverse data types. It doesn't necessitate explicit feature mapping knowledge, and it extends existing algorithms into non-linear domains.

9.2.1 Input Normalization

Input data in dataset D must be normalized for effective use of the kernel as a similarity measure. Several normalization techniques are commonly applied:

1. **Min-Max Normalization**: $\overline{x} = \frac{x - \min}{\max - \min}$ where min and max represent the minimum and maximum input values in D.
2. **Normalization** (Standardization): $\overline{x} = \frac{x - \mu}{\sigma}$ where μ is the mean and σ is the standard deviation of input values in D.

3. **Unit Vector Normalization**: $\overline{x} = \frac{x}{||x||}$

In the following discussion, **we assume the use of normalized input data** .

9.2.2 Kernel Types

- Linear: $k(x, x') = x^T x'$
- Polynomial: $k(x, x') = (\beta x^T x' + \gamma)^d, \quad d \in \{2, 3, \dots\}$
- Radial Basis Function (RBF): $k(x, x') = \exp\left(-\beta |x - x'|^2\right)$
- Sigmoid: $k(x, x') = \tanh(\beta x^T x' + \gamma)$

Recalling **SVMs** , we can use the kernel trick on that model, simplifying calculations.

9.2.3 Kernel Trick

Linear Classification with Kernel Trick

In SVM, the solution has the form: $w^* = \sum_{n=1}^{N} \alpha_n x_n$

- **Linear model (with linear kernel):** $y(x; \alpha) = \text{sign}\left(w_0 + \sum_{n=1}^{N} \alpha_n x_n^T x\right)$
- **Kernel trick:** $y(x; \alpha) = \text{sign}\left(w_0 + \sum_{n=1}^{N} \alpha_n k(x_n, x)\right)$

Linear Regression with Kernel Trick

Let's start considering the classical formulation of linear regression, for the model $y = w^T x$. We have to **minimize** the loss function $J(w) = \sum_{n=1}^{N} E(y_n, t_n) + \lambda ||w_k||^2$.

We can apply the **Kernel Trick** to get:

- $y(x; w^*) = \sum_{n=1}^{N} \alpha_n k(x_n, x)$
- $\alpha = (K + \lambda I_N)^{-1} t$

9.3 Kernel Methods: Summary

Kernel functions are mathematical functions that, when given two inputs, **yield a real value representing their similarity**.

$$k : X \times X \to \mathbb{R}$$

$$k(x, x') \text{ similarity of input samples } x \text{ and } x'$$

Linear Models with and without Kernel *Without Kernel:*

- **Model:** $y(x; \boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n x_n^T x$
- **Solution:** $\boldsymbol{\alpha} = (K + \lambda I_N)^{-1}\mathbf{t}$
- **Gram Matrix** K**:** $x_i^T x_j$ for $i, j = 1, \dots, N$

With Kernel:

- **Model:** $y(x; \boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n k(x_n, x)$
- **Solution:** Same as without kernel.
- **Gram Matrix** K**:** $k(x_i, x_j)$ for $i, j = 1, \dots, N$

The kernel trick replaces inner products $x^T x'$ with a kernel function $k(x, x')$, allowing for operations in high-dimensional spaces without the need for explicit **feature mappings** $\phi(x)$. This approach **simplifies the handling of non-linear patterns and extends algorithms** without requiring direct knowledge of $\phi(x)$, facilitating computations and supporting dimensionality reduction and non-linear analysis.

Normalization:
Min-Max: $\overline{x} = \frac{x - \min}{\max - \min}$
Standardization: $\overline{x} = \frac{x - \mu}{\sigma}$
Unit Vector: $\overline{x} = \frac{x}{||x||}$

In the following discussion, **we assume the use of normalized input data**
.

Kernel Types:
Linear: $k(x, x') = x^T x'$
Polynomial: $k(x, x') = (\beta x^T x' + \gamma)^d$
RBF: $k(x, x') = \exp(-\beta |x - x'|^2)$
Sigmoid: $k(x, x') = \tanh(\beta x^T x' + \gamma)$

SVM Kernel Trick:
Linear: $y(x; \alpha) = \text{sign}(w_0 + \sum \alpha_n x_n^T x)$
With Kernel: $y(x; \alpha) = \text{sign}(w_0 + \sum \alpha_n k(x_n, x))$

Linear Regression:
Minimize: $J(w) = \sum E(y_n, t_n) + \lambda ||w_k||^2$
Kernel Trick: $y(x; w^*) = \sum \alpha_n k(x_n, x)$, $\alpha = (K + \lambda I_N)^{-1} t$

Chapter 10

Instance Based Learning

Here we introduce non-parametric models. The core idea of **instance based learning**, is to make predictions without an explicit formulation of the output function we are trying to guess, with a **non-fixed** amount of parameters, this grows as the amount of data grows. Those are **non parametric modeks**, different from the models we saw up until this point.

10.1 K-NN Classification

- Finds the **K** nearest neighbours to the new point **x**
- Assigns the most common label among the **k** neighbours

The **overfitting** problem is reduced by increasing K but this means having memory storage of all the dataset.

To compare distances between points we can use this distance function, basically the likelihood of the class c for the new instance x:

$$p(c|x, D, K) = \frac{1}{K} \sum_{x_n \in N_K(x_n, D)} I(t_n = c)$$

- $p(c|x, D, K)$: Probability that a new data point x belongs to class c given the dataset D and considering K nearest neighbors.
- $\frac{1}{K}$: A normalization factor that divides the count by K, ensuring the result is a probability (i.e., between 0 and 1).
- $\sum_{x_n \in N_K(x_n, D)}$: Sum over the K nearest points to x_n within the dataset D. This sums the contributions of each of the K nearest neighbors to the probability.
- $N_K(x_n, D)$: The set of K nearest points to x_n in the dataset D. It represents the neighborhood of x_n defined by the K closest instances according to some distance metric (e.g., Euclidean distance).
- $I(t_n = c)$: An indicator function that returns 1 if the neighbor t_n's class is c (the condition $t_n = c$ is true), and 0 otherwise. This function is used to count how many of the K nearest neighbors belong to the class c.

$$||x - x_n||^2 = x^T x + x_n^T x_n - 2x^T x_n$$

This form can be **kernelized**

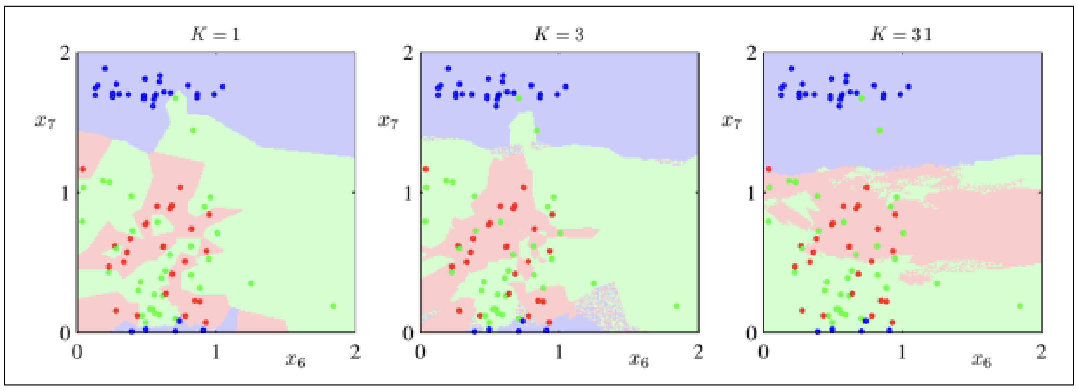


Figure 10.1. "In general, the bigger is k, the smoother the separation surface will be"

10.1.1 Locally Weighted regression

The concept of KNN can be extended also to the problem of regression. Given a dataset and a new instance, we can try to guess it by creating a local linear model that considers only the k-nearest points and make a prediction.

- Compute $N_K(x_q, D)$: Find the K-nearest neighbors of x_q within the dataset D.
- Fit a regression model $y(x; w)$ on $N_K(x_q, D)$: Use the K-nearest neighbors of x_q to fit a regression model, estimating the parameters w based on these neighbors.
- Return $y(x_q; w)$: Use the fitted regression model to predict the output y for the query point x_q, based on the learned parameters w.

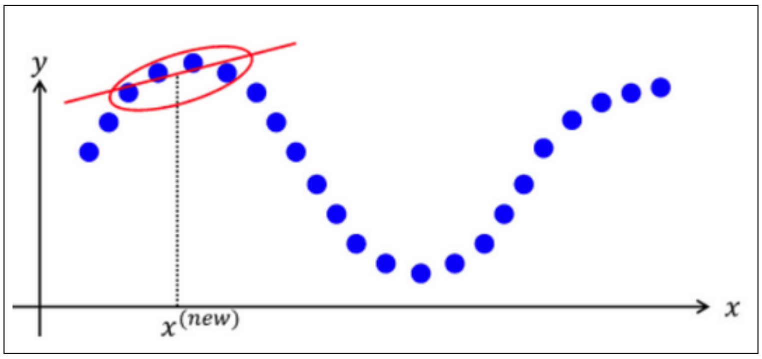


Figure 10.2. "Linear Kernel Example"

Chapter 11

Multiple Learners Learning

Ensemble learning is a strategy to enhance predictive accuracy by combining results from multiple models or learners, known as a "**committee** ." These models can be trained in two main ways:

11.1 Parallel Training

After, with the **same dataset** we train a set of $y_m(x)$ models.

- **Voting**, **regression** or **weighted majority (classification)** can be used to then determine the final output.

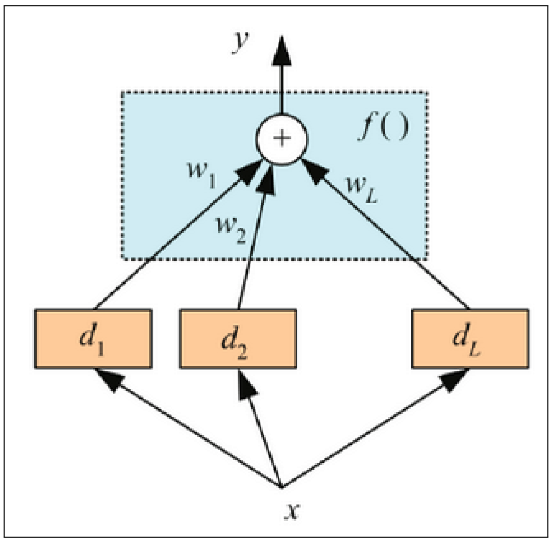


Figure 11.1. Voting

- **Mixture of Experts**, it implements a **gating mechanism** a learned function that dynamically routes input data to the most appropriate expert within the ensemble. It assigns weights to each expert, reflecting their relevance for the current input.

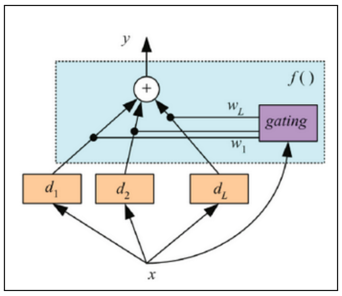


Figure 11.2. Mixture of Experts

- **Stacking**, a two-stage process, with base models trained on the dataset in the first stage and a meta-model trained on base model outputs in the second stage. Stacking is effective in diverse, real-world scenarios and helps mitigate overfitting.

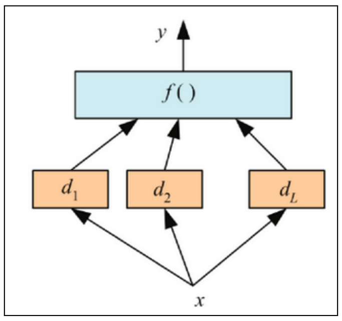


Figure 11.3. Stacking

- **Cascading**, cascading learners (based on confidence thresholds) is a technique that enhances classification accuracy by sequentially routing data through a series of classifiers, each with its own confidence threshold. It is especially useful for reducing false positives and improving decision-making efficiency in hierarchical classification systems.

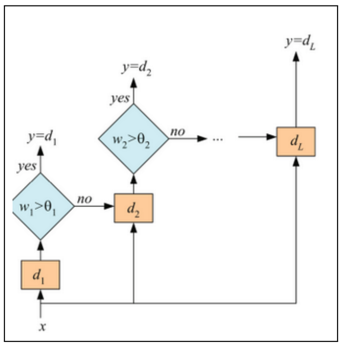


Figure 11.4. Cascading

- **Bagging**, uses subdivisions of the original dataset, different subsets are used to train different models, predictions are made using a **voting scheme**.

11.2 Sequential Training (Boosting)

The general approach of Boosting:

- Base classifiers (weak learners) trained sequentially
- Each classifier trained on weighted data
- Weights depend on performance of previous classifiers
- Points misclassified by previous classifiers are given greater weight
- Predictions based on **weighted majority of votes**

11.2.1 AdaBoost Algorithm

AdaBoost iteratively combines weighted weak learners. Initially, all data points have equal weights. In each iteration, a weak learner is trained on weighted data, and its weighted error rate is calculated. Based on the error, a weight (alpha) is assigned. Misclassified points get higher weights. Weights are normalized. The final strong classifier is a weighted sum of weak learners' predictions. AdaBoost adapts, focusing on challenging data points. The sign of the sum determines the class.

AdaBoost can be explained as the **sequential minimization** of an exponential error function.

Algorithm 4 AdaBoost

```
Require: D for Binary Classification,
  Initialize $w_n^{(1)} = 1/N$ for every n from 0 to N
  for m from 1 to M do
      Train  Weak learner minimize weight err. function:
          $$J_m = \sum_{n=1}^{N} w_n^{(m)} \cdot I(y_m(x_n) \neq t_n)$$
      Evaluate
          $$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} \cdot I(y_m(x_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}} \qquad \alpha_m = \ln\left(\frac{1-m}{m}\right)$$
      Update  the data weighting coefficients:
          $$w_n^{(m+1)} = w_n^{(m)} \cdot \exp\left(\alpha_m \cdot I(y_m(x_n) \neq t_n)\right)$$
  end for
  return $Y_M(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(x)\right)$
```

The objective is to minimize the error function E by adjusting α_m and $y_m(x)$ for $m = 1, \ldots, M$. Sequential minimization occurs step by step, with each iteration adjusting $y_M(x)$ and α_M. The explicit form of E shows that it depends on data points and weights. The weights $w_n^{(M)}$ are updated using a formula involving α_M, and α_M is calculated based on the weighted error rate of the weak learner.

The final prediction combines the weighted sum of $y_m(x)$ using the sign function. This process demonstrates that AdaBoost effectively minimizes the error function and is appreciated for its simplicity and versatility. However, it's sensitive to data quality and base learner accuracy, which can impact its performance.

11.3 Instance Based and Multiple Learners: Summary

Instance Based Learning
The core idea of **instance based learning**, is to make predictions without an explicit formulation of the output function we are trying to guess, with a **non-fixed** amount of parameters, this grows as the amount of data grows.
K-NN classification

- Finds the **K** nearest neighbours to the new point **x**
- Assigns the most common label among the **k** neighbours

To compare distances between points we can use this distance function, basically the likelihood of the class c for the new instance x:

$$p(c|x, D, K) = \frac{1}{K} \sum_{x_n \in N_K(x_n, D)} I(t_n = c)$$

K-NN for Regression (Locally Weighted regression)
The concept of KNN can be extended also to the problem of regression. Given a dataset and a new instance, we can try to guess it by creating a local linear model that considers only the k-nearest points and make a prediction.

- Compute $N_K(x_q, D)$: Find the K-nearest neighbors of x_q in D.
- Fit a regression model $y(x; w)$ on $N_K(x_q, D)$: Use the K-nearest neighbors of x_q to fit a regression model, estimating w based on neighbors.
- Return $y(x_q; w)$: Use the fitted regression model to predict the output y for the query point x_q, based on the learned parameters w.

Multiple Learners Parallel Training

- **Voting:** Determines final output through collective decision-making methods.
- **Mix of Experts:** Gating mechanism, route inputs to best expert with weights.
- **Stacking:** Employs a two-stage model training process, using base and meta-models to enhance performance and reduce overfitting.
- **Cascading:** Improves accuracy by passing data through multiple classifiers based on confidence levels, reducing false positives.
- **Bagging:** Trains models on different subsets, voting scheme for predictions.

Sequential Training (Boosting) AdaBoost AdaBoost targets difficult data points across iterations by initially treating all data equally. It trains simple models, emphasizes errors by adjusting data point weights, and iterates to correct prior misclassifications. The culmination is a composite model where each simple model contributes based on its accuracy. This method progressively refines the classifier, enhancing its ability to tackle complex patterns through a series of corrective steps.

- Start with equal weights for all data points.
- Iteratively:
  - Train a weak model on current weighted data.
  - Calculate model error and influence (weight)
  - Increase weight of misclassified data, focusing subseq. models on these points.
- Aggregate the models, weighting votes by their accuracy, to form the final classifier.

Chapter 12

Artificial Neural Networks

12.1 Introduction

As with other Machine Learning techniques, the goal of Neural Networks is to find a function that best approximates a target function. The core approach involves defining a function with inputs x (the input of our problems) and w (the weight of our model). To evaluate our model's predictions, we define an **error function** and minimize it by tuning the parameters.

Unlike other classical machine learning techniques, neural networks use **non-linear functions** with respect to both the input space and the model's parameters. This non-linearity is a key distinction, making the model non-linear in its parameters.

12.1.1 The Perceptron

The intuition behind neural networks is derived from biological neural networks. At its core, a neural network consists of units called **perceptrons** . A perceptron is a mathematical abstraction that operates as follows: given inputs and weights, it computes a **linear combination** and then passes this through an **activation function** to produce the output.

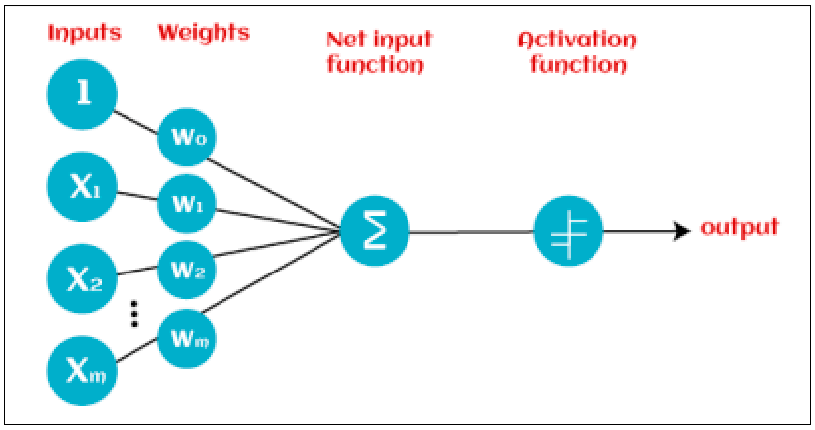


Figure 12.1. Perceptron Visualized

12.2 Artificial Neural Networks (ANNs)

12.2.1 ANNs - Definitions and Purpose

ANNs, also known as Neural Networks (NN), Feedforward Neural Networks (FNN), or Multilayer Perceptrons (MLP), serve as function approximators using a parametric model. **They are well-suited for tasks involving associating one vector with another.**

The concept practical idea is realizing a **network of interconnected perceptrons**

The goal of ANNs is estimate a function $f : X \to Y$, where Y can be a set of classes or a continuous range. The data for ANNs is given as $D = \{(x_n, t_n)\}_{n=1}^{N}$, with the aim that $f(x_n)$ approximately equals t_n. The framework defines $y = \hat{f}(x; \theta)$ and focuses on learning the parameters θ such that $\hat{f}$ approximates f.

12.2.2 Feedforward Networks

Inspired by brain structures, feedforward networks comprise hidden layers where each unit's (neuron's) activation depends on its connections with previous units. However, these networks are not precise models of the brain but use some insights from its structure.

In FNNs, information flows **from input to output** without loops. The network function f is a composition of elementary functions in an **acyclic** graph, where for example $f(x; \theta) = f^{(3)}(f^{(2)}(f^{(1)}(x; \theta^{(1)}); \theta^{(2)}); \theta^{(3)})$, with $f^{(m)}$ being the m-th layer and $\theta^{(m)}$ its parameters. These networks are **chain structures without loops** , where the **length** of the chain determines the depth. **Deep learning involves networks with a significant number of layers.**

FNNs **are preferred because linear models cannot adequately model interactions between input variables**. While **kernel methods** necessitate the choice of suitable kernels (like RBF or polynomial), FNNs learn complex combinations of many parametric functions, presenting a non-convex problem.

12.2.3 XOR Example

We want to train a neural network on the representation of the XOR function

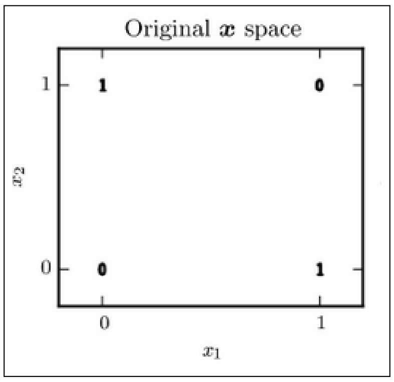


Figure 12.2. XOR function

We can start defining a Neural Network with this simple structure:

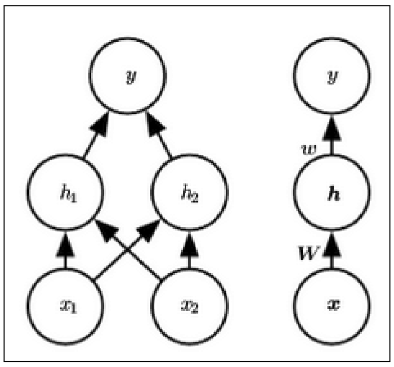


Figure 12.3. Simple two layered Network

Activation functions define neuron outputs. Typically, ReLu for hidden layers, identity for output layer. ReLu makes negatives 0, keeps positives.

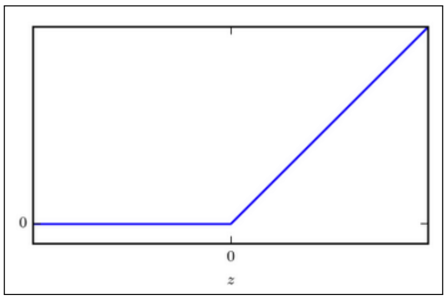


Figure 12.4. $g(\alpha) = \max(0, \alpha)$

Basically, every neuron is connected to the whole input vector and each neuron has its own **weights** These are the computation that will be performed

$$
\begin{aligned}
h_1 &= \text{ReLu}\left(\begin{bmatrix} w_{11} & w_{12} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + w_{10}\right) \\
h_2 &= \text{ReLu}\left(\begin{bmatrix} w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + w_{20}\right) \\
y &= \begin{bmatrix} w_{h1} & w_{h2} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + b = \mathbf{W}^T \mathbf{h} + b
\end{aligned}
$$

We can chose the **Mean Squared Error** as our error funcion.

Universal approximation theorem

A feed forward neural network with one linear output layer and at least one hidden layer with any "squashing" function is capable of approximating any **Borel measurable function** with any desired amount of error, provided that enough hidden units are used.

12.3 Defininig The Cost Fucntion

We define the cost function based on the concept of **likelihood**. In particular, we are interested in the probability of obtaining the correct prediction given the input x and the weights of the neural network: $P(t \mid x, \theta)$. The error is defined as the negative **log-likelihood**:

$$J(\theta) = \mathbb{E}_{x,t \in D}[-\ln(P(T \mid x, \theta))]$$

Similarly to the linear model case, if we assume assuming that the likelihood's distribution is modeled by some Gaussian noise:

$$J(\theta) = \mathbb{E}_{x,t \in D}\left(\frac{1}{2}||t - f(x;\theta)||^2\right)$$

Where $||t - f(x;\theta)||^2$ is **the mean squared error**, since finding the maximum likelihood estimation with Gaussian noise corresponds to minimizing the **mean squared error**.

We choose the proper cost function by identifying the nature of our problem: **Regression, Binary classification, Multi-class classification.**

12.3.1 Regression

- **Output layer**, linear model with **identity activaction function** $y = W^T h + b$
- **Loss function**, **mean squared error**
  (if we assume a Gaussian noise distribution $P(t \mid x) = \mathcal{N}(t \mid \gamma, \beta^{-1})$)

12.3.2 Binary Classification

- **Output layer**, linear model with **sigmoid activaction function** $y = \sigma(W^T h + b)$
- **Loss function**, **Softplus** (It can be viewed as a smooth version of ReLU)

$$-\ln(P(t \mid x)) = -\ln[\sigma((\alpha)t(1-\alpha)^{1-t})] = -\ln[\sigma((2t-1)-\alpha)] = \text{softplus}((1-2t)\alpha)$$

- Remember that $\alpha = W^T X + b$.

12.3.3 Multi-class Classification

- **Output layer**, linear model with **Softmax activaction function**

$$\text{softmax}(\alpha_i) = \frac{e^{\alpha_i}}{\sum_j e^{\alpha_j}}$$

(Softmax normalizes output for each class. Best prediction will correspond to class with the closest value to 1)

- **Loss function**, **categorical cross-entropy**

$$J_i(\theta) = \mathbb{E}_{x,t\sim D}\left[-\ln \text{softmax}(\alpha^{(i)})\right]$$

(likelihood is a multinomial distribution)

- Remember that $\alpha^(i) = W_i^T X + b$.

12.4 Computing Gradient and Activation Functions

12.4.1 Activation Functions

The premise for the choice of the **activation function** is that there is **no theoretical better one** . Empirically, it has been shown that ReLus are the best choice expecially if we don't have any particular information about the dataset. Being non-linear function makes them robust to the phenomenon of flat regions. Their only drawback is that ReLus **are not differential in 0**, but in practice this won't be a problem. The formal definition of a ReLu is:

$$g(\alpha) = \max(0, \alpha)$$

12.4.2 Gradient

When implementing a neural network it's fundamental to choose an efficient way for **computing the gradient**. The default algorithm is **Backpropagation** .

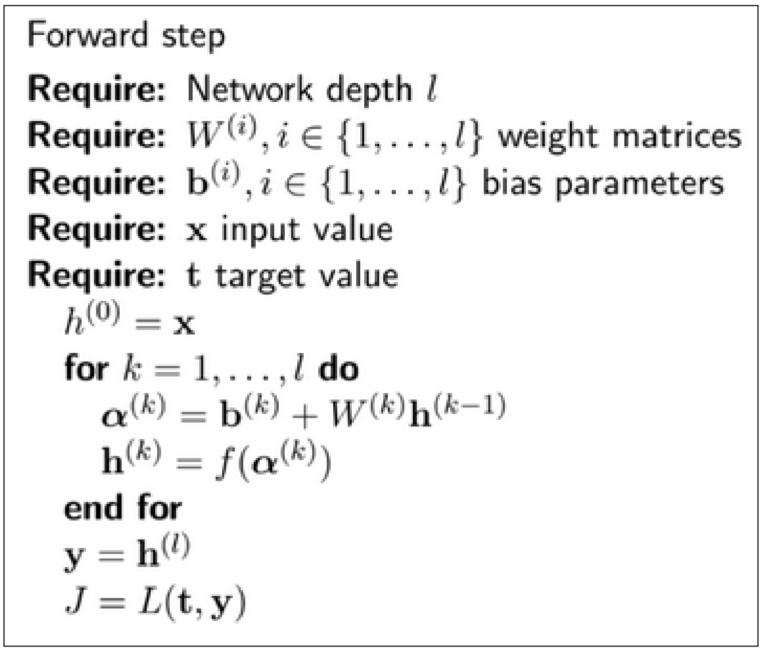
The general idea for backpropagation is having an **iterative approach**, where each iteration has two steps:

- **Forward Step**: compute output value for each layer until we get to the **output**

- **Backward Step**: compute loss between network's output and the corrisponding target value and adjusts the values for each layer based on the gradient.

In order to get to the actual computation of the gradient we must start by noticing that our final output is the result of a **composition of function**. Thus, the gradient for the loss function will be computed exploiting the **chain rule** for **derivatives**

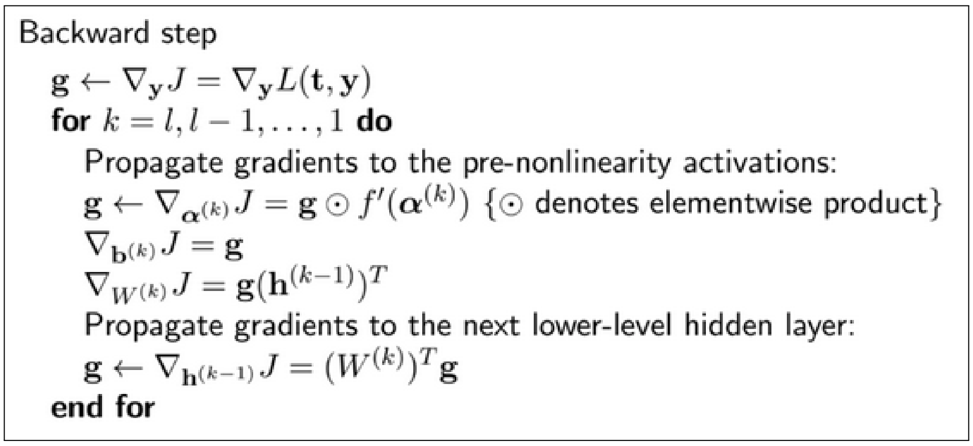
Forward Step



```
Forward step
Require: Network depth $l$
Require: $W^{(i)}, i \in \{1, \dots, l\}$ weight matrices
Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$ bias parameters
Require: $\mathbf{x}$ input value
Require: $\mathbf{t}$ target value
  $h^{(0)} = \mathbf{x}$
  for $k = 1, \dots, l$ do
    $\boldsymbol{\alpha}^{(k)} = \mathbf{b}^{(k)} + W^{(k)}\mathbf{h}^{(k-1)}$
    $\mathbf{h}^{(k)} = f(\boldsymbol{\alpha}^{(k)})$
  end for
  $\mathbf{y} = \mathbf{h}^{(l)}$
  $J = L(\mathbf{t}, \mathbf{y})$
```

Figure 12.5. Forward Step Pseudocode

Backward Step



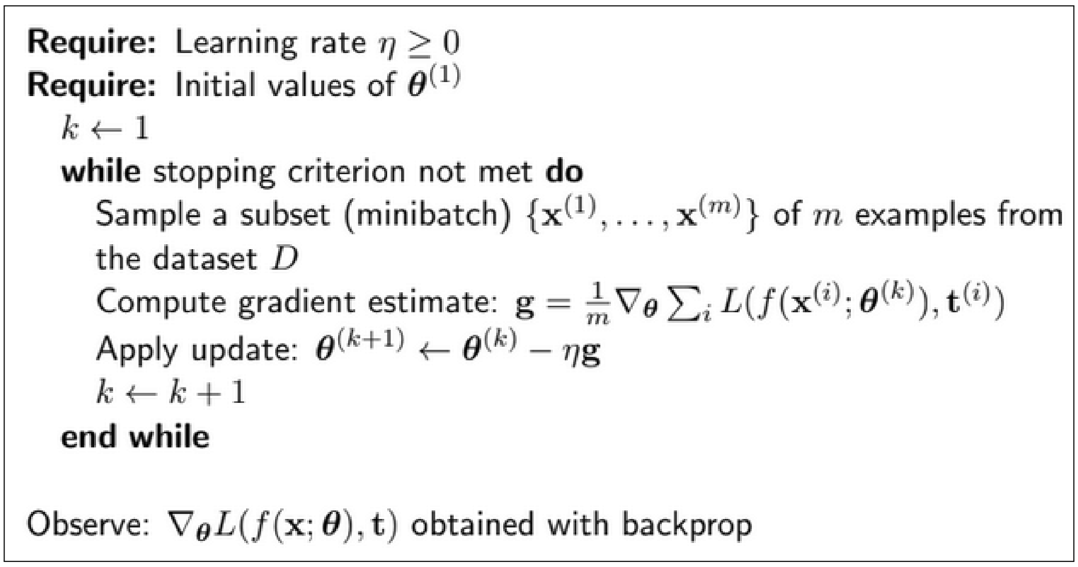
```
Backward step
  $\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{t}, \mathbf{y})$
  for $k = l, l-1, \dots, 1$ do
    Propagate gradients to the pre-nonlinearity activations:
    $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(k)}} J = \mathbf{g} \odot f'(\boldsymbol{\alpha}^{(k)})$ {$\odot$ denotes elementwise product}
    $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$
    $\nabla_{W^{(k)}} J = \mathbf{g}(\mathbf{h}^{(k-1)})^T$
    Propagate gradients to the next lower-level hidden layer:
    $\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = (W^{(k)})^T \mathbf{g}$
  end for
```

Figure 12.6. Backward Step Pseudocode

Rember that ultimately our goal is to **compute the gradient of the loss function with respect to the weights and biases of each layer** , in order to tune them properly and **lower the error at the next iteration**. Notice that the backpropagation algorithm will be repeated many times during the training of our network. This results in a serious problem from a computational cost standpoint, meaning that even if this technique works in theory, **in practice we might never reach a result in reasonable time**.

12.4.3 Stochastic Gradient Descent

In practice, as input size grows, backpropagation becomes computationally demanding. Stochastic gradient descent offers a solution by computing **backpropagation for mini-batches** (random subsets of size 'm' from the training set) in each iteration. Loss is calculated per mini-batch, and then averaged across mini-batches.



```
Require: Learning rate $\eta \geq 0$
Require: Initial values of $\boldsymbol{\theta}^{(1)}$
  $k \leftarrow 1$
  while stopping criterion not met do
    Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ of $m$ examples from the dataset $D$
    Compute gradient estimate: $\mathbf{g} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$
    Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - \eta \mathbf{g}$
    $k \leftarrow k + 1$
  end while

Observe: $\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{t})$ obtained with backprop
```

Figure 12.7. Stochastic Gradient Descent

In the algorithm, we encounter the term η is the **learning rate**, which governs the step size in the gradient direction. Selecting its value poses two key challenges. If it's too large, we risk overshooting the optimum, causing oscillations and divergence. Conversely, if too small, progress is sluggish, potentially never reaching the local optimum.

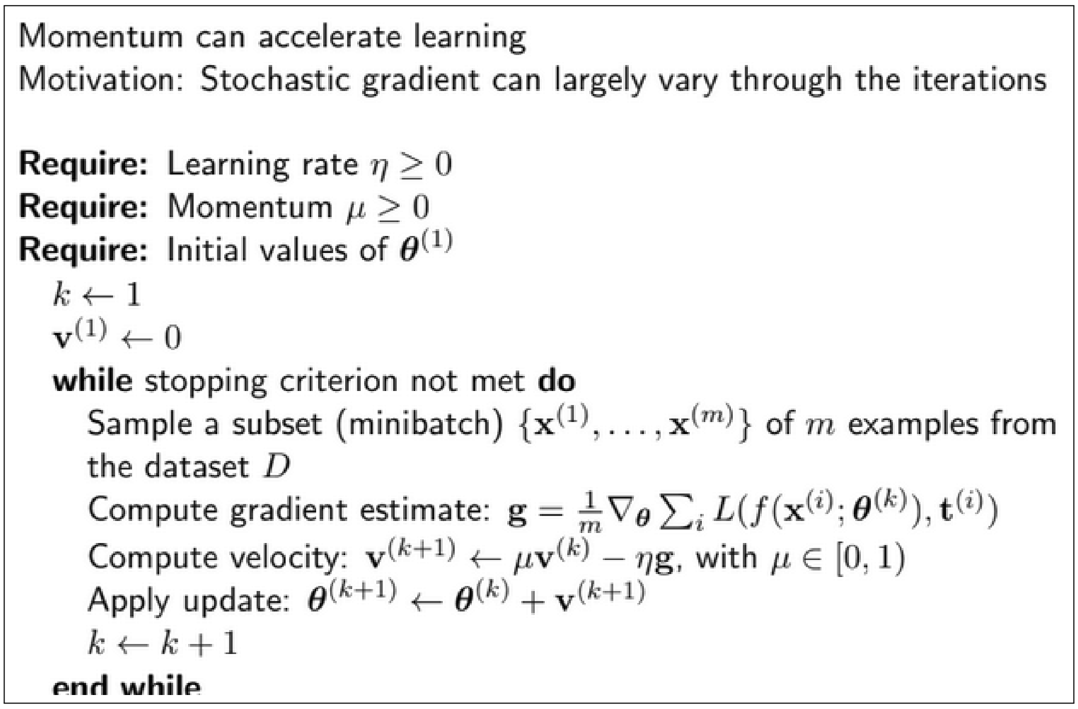
Classical vs. stochastic gradient descent comparison shows both can find an optimum, but stochastic gradient descent requires more steps. Crucially, each SGD step is significantly faster.

A possible solution to properly tune the learning rate is an adaptive approach. We start from a big value of and we keep decreasing it at each iteration, until a certain step. This is a clever approach, since it's more likely that we start far from the optimum, and then get closer to the solution as the number of iteration increases.

12.4.4 SGD with momentum

The intuition for this approach comes from the physics concept of momentum and potential energy. When we let a ball roll from a cliff, it will still have a tendency to move forward even when it reached a plain surface due to the accumulation of momentum. The same concept can be applied to the computation of the gradient.

It introduces a **second parameter** other than the learning rate, will regulate how much we will keep moving even if we are still. One notable technique is **Nesterov momentum**, a technique that uses the same concept of momentum, but chooses to apply momentum before the computation of the gradient (sometimes it works better).



Momentum can accelerate learning
Motivation: Stochastic gradient can largely vary through the iterations

```
Require: Learning rate $\eta \geq 0$
Require: Momentum $\mu \geq 0$
Require: Initial values of $\boldsymbol{\theta}^{(1)}$
  $k \leftarrow 1$
  $\mathbf{v}^{(1)} \leftarrow 0$
  while stopping criterion not met do
    Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ of $m$ examples from the dataset $D$
    Compute gradient estimate: $\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$
    Compute velocity: $\mathbf{v}^{(k+1)} \leftarrow \mu \mathbf{v}^{(k)} - \eta \mathbf{g}$, with $\mu \in [0, 1)$
    Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} + \mathbf{v}^{(k+1)}$
    $k \leftarrow k + 1$
  end while
```

Figure 12.8. Stochastic Gradient Descent

We have seen how to compute the gradient of the loss with respect to each parameter of the network. Optimizers are the specific function that take as an input the loss function and use it to tune the parameters. Popular optimizers are: AdaGrad, RMSProp, Adam. . .

We have seen how in theory increasing the size of a neural network or the number of training iteration makes our model more performant. In reality we have to take into account that during training we are fitting our network just on a **subset of samples**. This will cause to hit a certain point where while the error on the training keeps decreasing, the error on the test set will start increasing. When this happens, we are facing **overfitting** .

12.5 Regularization Methods

12.5.1 Parameter Norm Penalties

Overfitting can be caused by excessively high parameter values in a model. To address this, we add a regularization term to the cost function that penalizes parameters with high values. The resulting cost function will be:

$$J(\theta) = \text{Original cost function}$$

Where λ is the regularization term's coefficient, tuning its effect on regularization.

12.5.2 Dataset Augmentation

Overfitting may occur when a model fits the training data too precisely, resulting in poor generalization. To mitigate this, introduce variations in the training data by adding noise or producing samples with small changes such as image rotation, scaling, or illumination adjustments.

12.5.3 Early Stopping

Monitor the performance of the network during training by comparing train and test loss. Stop training when the train loss continues to decrease, but the test loss starts increasing or remains stable.

12.5.4 Parameters Sharing

A large number of parameters can improve a network's performance but increases the risk of overfitting. To combat this, constrain the number of parameters by sharing some values. Only a subset of parameters (trainable parameters) is updated during each iteration, while others remain shared.

12.5.5 Dropout

Like parameters sharing, dropout is a method to manage large parameter counts. The concept involves maintaining the same network structure but randomly selecting a certain percentage of connections to be trained while others remain unchanged.

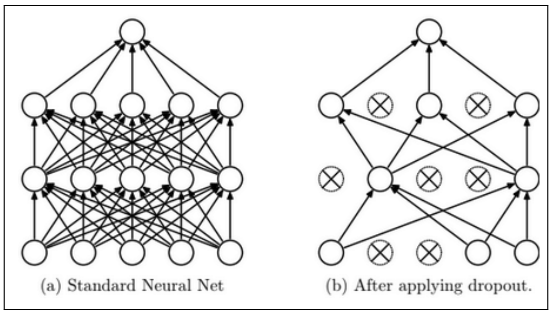


Figure 12.9. Dropout Example

12.6 ANN: Summary

Neural Networks consist of units called **Perceptrons** , computing outputs from inputs and weights through **linear combinations** and **activation functions** . They excel as function approximators, associating vectors. **Feedforward networks**, with **acyclic graphs** and **hidden layers**, flow information linearly from input to output. Activation functions like **ReLu** (for hidden layers) and identity (for outputs) dictate neuron outputs. The **cost function** is defined by the negative log-likelihood, focusing on the probability of correct predictions given inputs and network weights.

Cost and activation function selection depends on the problem type, **there's no universally superior activation function** .

- Regression (Activation: identity, Loss: Mean Squared Error)
- Binary Classification (Activation: Sigmoid, Loss: Softplus)
- N-Class Classification (Activation: Softmax, Loss: Categorical Cross-Entropy)

Computing the gradient

Backpropagation: The general idea for backpropagation is having an iterative approach, where each iteration has two steps:

- **Forward Step**: compute output value for each layer until we get to the output
- **Backward Step**: compute loss between network's output and the corrisponding target value and adjusts the values for each layer based on the gradient

Backpropagation's computational load increases with input size.
Stochastic gradient descent (SGD) addresses this by using **mini-batches**, calculating and averaging loss per mini-batch. The **learning rate** (η) affects step size towards the optimum; too large causes oscillations, too small slows progress. While both classical and stochastic gradient descent can find optima, SGD is faster per step but may require more steps.**An adaptive learning rate strategy**, starting high and decreasing over iterations, **optimizes convergence by accommodating proximity to the solution**. **Momentum** in gradient computation **mimics physical momentum**, enabling continued movement even without new force, improving optimization. Nesterov momentum adjusts gradients with **preemptive momentum**.

Chapter 13

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are networks that exploit the operation of convolution in order to process high dimensional data (like images) **transforming the dimensionality of the original space**

So, when we talk about "transforming the dimensionality," we mean that the input data, which starts as a high-dimensional grid (e.g., an image), goes through a series of operations (convolutions, pooling, flattening) that progressively reduce its spatial dimensions and transform it into a format that is suitable for making predictions or classifications.

13.1 Convolutions

Convolution is a mathematical operation performed between two functions, typically an input function I and a kernel function K, in mathematics is a way of merging two functions into a single function that represents how one function modifies the shape of another.

- *Continuous Convolution:*

$$(I * K)(t) = \int_{-\infty}^{\infty} I(a)K(t-a)\,da$$

- *Discrete Convolution:*

$$(I * K)(t) = \sum_{a=-\infty}^{\infty} I(a)K(t-a)$$

Convolutions in 2D and 3D

In the n-dimensional convolution the **kernel** moves along n-dimensions.

- *2D Convolution:*

$$(I * K)(i,j) = \sum_{m \in S_1} \sum_{n \in S_2} I(m,n)K(i-m, j-n)$$

- *3D Convolution:*

$$(I * K)(i, j, k) = \sum_{m \in S_1} \sum_{n \in S_2} \sum_{u \in S_3} I(m, n, u) K(i - m, j - n, k - u)$$

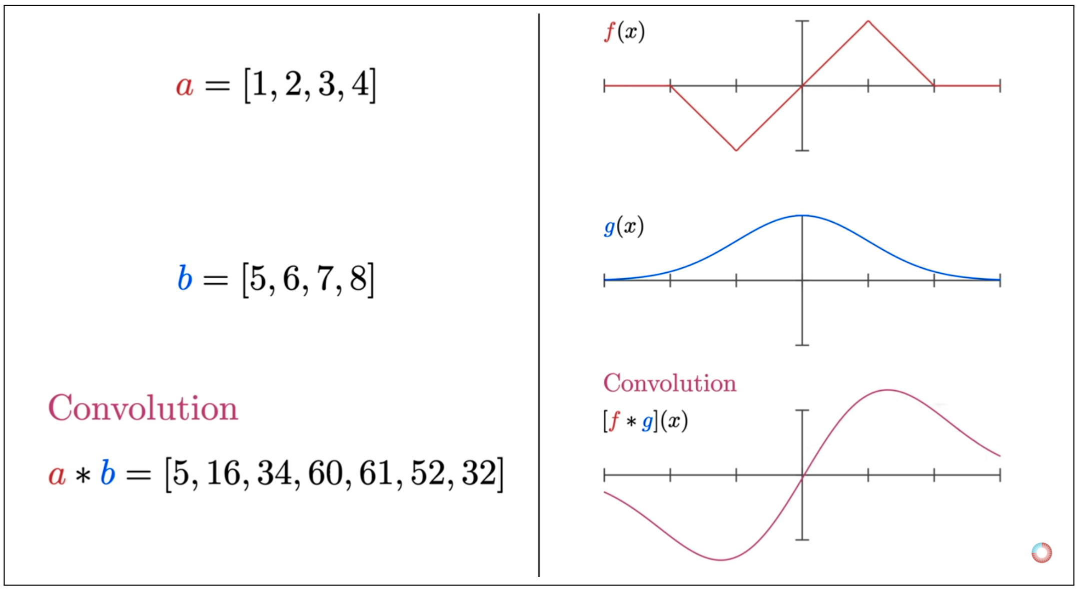


Figure 13.1. Convolution example (from 3blue1brown video

13.1.1 Properties of Convolutions

- *Commutative:* $(I * K)(i, j) = (K * I)(i, j)$
- *Cross-correlation:* Flipping the kernels in the convolution operation does not change the result.

Our goal with convolution will be finding the proper kernels for our problem. We can think about kernels as **filters that can be applied to inputs** through the operation of convolution.

The **dimension of the convolution defines the direction where the kernel will move**. In practice, when we apply convolution each time we take a slice of the input corresponding to the size of the kernel, we multiply element-wise the two slices and sum each element. Then the kernel is shifted and the operation is repeated at each shift.

A typical architecture for a CNN consists in **three layers**: an input layer, a **convolutional layer** and finally an output layer, we define the feature map (or depth slice) a particular section of the output. In general, we can say that the output of a convolution is a composition of feature maps.

The trainable parameters of a convolutional layer correspond to the total parameters of its kernels. The i-th convolutional layer will have trainable parameters.

13.2 Convolutional Layers

Every convolutional lauyer is composed of three stages:

- **Convolutional stage**, convolutes the input with a kernel.
- **Detector stage**, applies a non-linear **activation** function.
- **Pooling stage**, will transform the output of the detector by applying another filter.

When defining the architecture of a CNN, we have to remember the problem we want to solve, which is usually to work with images or videos. In these cases we care about **locality** where considering the features of the input. In fact it's much more likely that a pixel of an image is correlated to it's neighbour with respect to another pixel far away.

This is why introduce the concept of **sparse connectivity** . Instead of working with fully connected layers, we exploit the concept of locality, we connect a pixel just to the adjacent one, thus saving in terms of numbers of parameters needed.

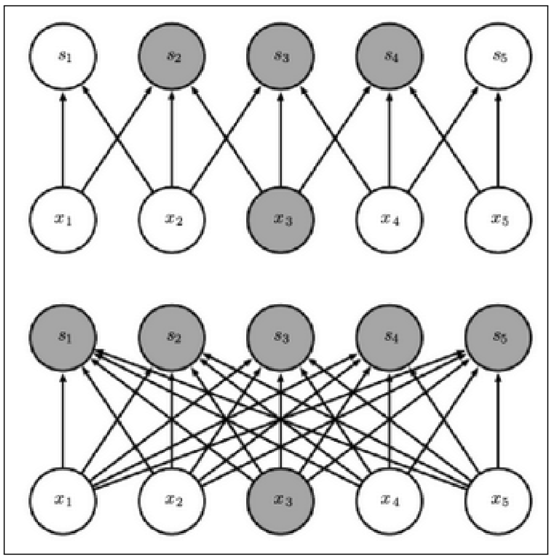


Figure 13.2. Sparse Connectivity vs fully connected layer

Parameter sharing is a constraint on the weights imposing that some of the weights of the same layer must have the same value. This method allows for a substantial reduction in the number of total trainable parameters. When we use a kernel, we are basically **repeating operations on different slices of the output by using each time the same kernel parameters**. In fact, when we look at how kernels work, they basically implement both concepts of sparse connectivity and parameter sharing.

We have seen that in general the operation of convolution between an input and a kernel results in an output with lower dimensions that the input, notice that this happens if the convolution operation starts with the kernel aligned to the border of the input.Sometimes, we may want our output to have the same size of the input.

In this cases we introduce the concept of padding. Basically we compensate the loss of size by extending the origin input using some filling values (usually 0).

13.2.1 Pooling Stage

The pooling stage is responsible for further decreasing the size of the final output byapplying a filtering operation. This time, the filter consists in a **well defined function** that has no trainable parameters. Two common approaches for the pooling stage are **max-pooling** (we extract the maximum value from the considered region), **average-pooling** (returns the average value from the considered region)

*An important parameter for padding is **stride**. Stride defines by how much the pooling filter moves at each step. In case of more than one dimension, we can define different stride parameters for each dimension. Pooling is considered as a subsampling operation*

13.2.2 Size of the Output (Feature Map)

Consider input of size $w_{\text{in}} \times h_{\text{in}} \times d_{\text{in}}$, d_{out} kernels of size $w_k \times h_k \times d_{\text{in}}$, stride s, and padding p. Dimensions of the output feature map are given by:

$$w_{\text{out}} = \frac{w_{\text{in}} - w_k + 2p}{s} + 1$$

$$h_{\text{out}} = \frac{h_{\text{in}} - h_k + 2p}{s} + 1$$

Number of trainable parameters of the convolutional layer is:

$$|\theta| = w_k \cdot h_k \cdot d_{\text{in}} \cdot d_{\text{out}} + d_{\text{out}}$$

13.2.3 Transfer Learning

The main idea with transfer learning is to use pre-trained model on new examples belonging to a similar domain. A major advantage of this method is to reduce the computation needed for training from scratch a new model.

If both source and target domain are similar (they come from a similar distribution), we can think about keeping the feature extraction part the same and re-training only the last layers. This approach is called fine-tuning. Notice that this method still requires a lot of computation, since the final layers of a CNN are dense.

Another intelligent approach is to exploit our CNNs just as feature extractors. In fact, we have seen that CNNs are really good at extracting the fundamental features of the input via the operation of convolution. We could use this property in order to create a pipeline where the CNN extracts the features, that are then fed to another ML model in order to compute a prediction.

13.3 CNN: Summary

Convolution merges two functions (input I and kernel K) into one, showing how one modifies the other's shape. Practically, it involves **element-wise multiplication** of input slices and the kernel, summing them up, then shifting the kernel and repeating.

$$(I * K)(t) = \int_{-\infty}^{\infty} I(a)K(t-a)\,da$$

A typical CNN architecture includes an *input layer*, *convolutional layer*, and *output layer*, producing a composition of feature maps as output.
Every convolutional lauyer is composed of three stages:

- **Convolutional stage**, convolutes the input with a kernel.
- **Detector stage**, applies a non-linear **activation** function.
- **Pooling stage**, will transform the output of the detector by applying another filter.

CNN architecture design considers the problem, often image or video processing, emphasizing **locality** and adjacent pixel correlation. **Sparse connectivity** reduces full connections to adjacent ones, saving parameters. **Parameter sharing**, where layer weights are constrained to equal values, cuts total trainable parameters by reusing kernel operations across output slices, embodying sparse connectivity and parameter sharing. Convolution typically reduces input dimensions unless padding is used to maintain original size by extending input edges with filler (usually 0).
Consider input of size $w_{\text{in}} \times h_{\text{in}} \times d_{\text{in}}$, d_{out} kernels of size $w_k \times h_k \times d_{\text{in}}$, stride s, and padding p. Dimensions of the output feature map are given by:

$$w_{\text{out}} = \frac{w_{\text{in}} - w_k + 2p}{s} + 1$$

$$h_{\text{out}} = \frac{h_{\text{in}} - h_k + 2p}{s} + 1$$

Number of trainable parameters of the convolutional layer is:

$$|\theta| = w_k \cdot h_k \cdot d_{\text{in}} \cdot d_{\text{out}} + d_{\text{out}}$$

Transfer learning leverages pre-trained models for new, similar domain tasks, cutting down on training time and computation. Fine-tuning, where the feature extraction layers are kept while re-training the last dense layers, suits similar source and target domains but is computationally intensive. Alternatively, using CNNs as **feature extractors** for another ML model to compute predictions is an efficient method to exploit CNNs' convolutional feature extraction capabilities

Chapter 14

Unsupervised Learning

The machine learning techniques that we studied in the previous lessons were about **supervised learning**, meaning that we assumed that our training datasets were labelled.

When dealing with real problems, many times we have to face situations were we only have unlabelled or partially labelled samples, this problems belong to the discipline of **unsupervised learning** .

14.1 Gaussian Mixture Models

The core idea behind Gaussian Mixture Models (GMM) is that we assume that our data was generated from a certain probability distribution expressed as a weighted sum of k Gaussians:

$$P(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

- $P(x)$ is the probability density function.
- π_k represents the **prior probability**.
- $\mathcal{N}(x; \mu_k, \Sigma_k)$ is the probability density function of a Gaussian distribution with mean μ_k and covariance Σ_k.

Given all of these parameters, it's easy to generate a new dataset by sampling data points from the distribution, once we have generated our data, we can forget about their parameters of the original distribution and try to learn them by starting from scratch, given that the number of distributions k is known.

Each instance x_n is generated by:

- Choosing Gaussian k according to prior probabilities $[\pi_1, \ldots, \pi_K]$
- Generating an instance at random according to that Gaussian, thus using μ_k, Σ_k

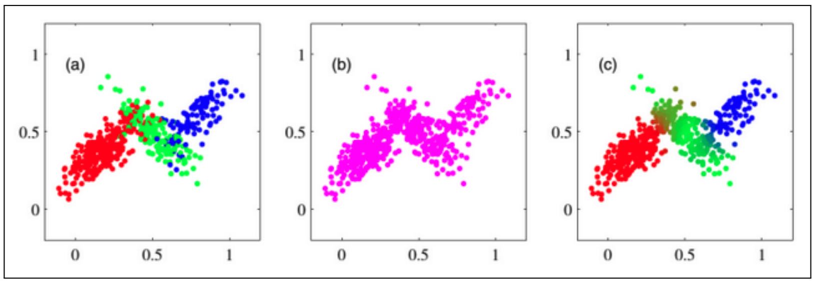


Figure 14.1. Gaussian Mixture Model Example

14.2 K-means Clustering

The K-means algorithm is a method used to group data into clusters. The "K" in K-means refers to the number of clusters you want to create.

1. **Initialization:** Choose the number of clusters k.
2. **Initial Partition:**
   (a) Start with the first k samples as single-element clusters.
   (b) For the remaining samples, assign each to the nearest centroid cluster and update the centroid.
3. **Assignment:** Assign each sample to the cluster with the closest centroid. If a sample's assignment changes, update the involved centroids.
4. **Convergence:** Repeat the assignment step until no further changes occur in the clusters.

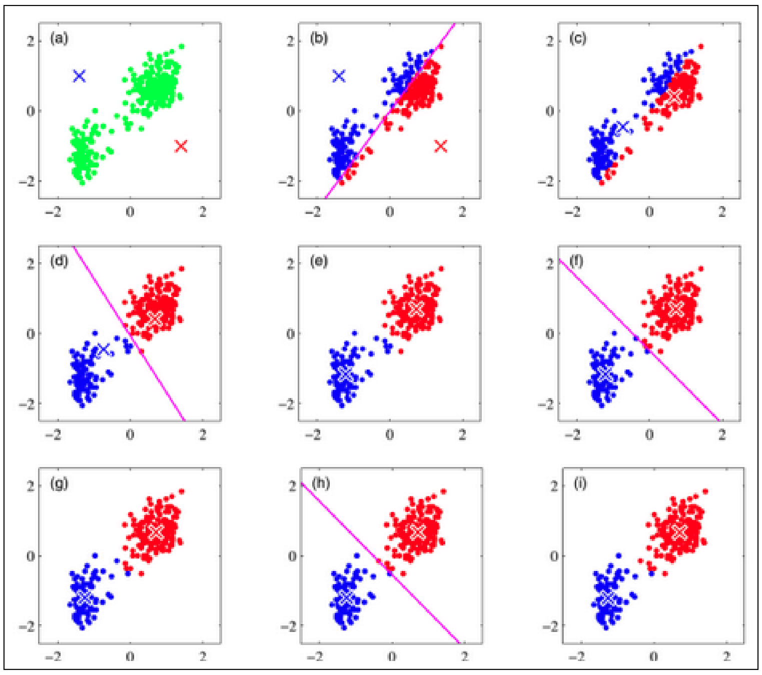


Figure 14.2. K-Means clustering example

14.2.1 Convergence of K-means

Convergence is guaranteed if:

- Every reassignment reduces the sum of distances within clusters.
- The number of possible partitions is finite.

14.2.2 About K-Means

CONS:

- The number of clusters K **must be determined beforehand**.
- It is sensitive to the initial conditions, which can lead to **local optima** when there are few data points available.
- K-means is not robust to outliers. Data points far from the centroid may pull the centroid away from the real one.
- **The resulting clusters tend to be circular in shape because K-means relies on distance metrics**.

Some solutions:

- Use K-means clustering only if there is a **substantial amount of data** available.
- Consider using the median instead of the mean for centering clusters.
- Define better distance functions that can handle outliers and non-circular clusters.

14.3 Latent Variables

Latent variables are a special kind of variables that are useful to describe our problem, **but are not given by the dataset**. They help us in better defining the formulation of our problem. When we analyze samples we don't know from which of the k distributions they come from. Hence, we can think about introducing the boolean variables associated with everyone of the k distributions (1 out of k approach for every vector) let's define:

$$P(z_k = 1) = \pi_k$$

thus

$$P(z) = \prod_{k=1}^{K} \pi_k^{z_k} \quad (z_k = 1 \text{ only for one value of } k, 0 \text{ otherwise})$$

(the likelihood of that vector is given by the products of the **posteriors elevated to the value of the latent variable**. Notice that **only one** latent varable has value 1). For a given value of z:

$$P(x|z_k = 1) = \mathcal{N}(x; \mu_k, \Sigma_k)$$

Notice that if we know Z, then the probability only depends on the k-th Gaussian. Given the previous considerations, we can express the posterior as:

$$P(x|z) = \prod_{k=1}^{K} \mathcal{N}(x; \mu_k, \Sigma_k)^{z_k}$$

Joint distribution:

$$P(x, z) = P(x|z)P(z) \quad \text{(chain rule)}$$

We obtained the formulation of a Gaussian Mixture Model. A fundamental assumption that we can derive is that a Gaussian Mixture Model can be expressed as the **marginalization of the joint probabilities** . Our problem now is that we don't actually know the value of the latent variables. We can now define the posterior probability, meaning the probability that a certain data comes from a certain distribution as:

Let's define the posterior

$$\gamma(z_k) \equiv P(z_k = 1|x) = \frac{P(z_k = 1)P(x|z_k = 1)}{P(x)}$$

$$\gamma(z_k) = \frac{\pi_k \mathcal{N}(x; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x; \mu_j, \Sigma_j)}$$

Note:

- π_k: prior probability of z_k
- $\gamma(z_k)$: posterior probability after observation of x.
- $\sum_{j=1}^{K} \pi_j \mathcal{N}(x; \mu_j, \Sigma_j)$ is the **normalization layer**

14.4 Expectation Maximization

While before we started with the assumption that both **covariance** and **prior** are the **same for every distribution**, now we are interested in **determing all the three parameters**. In particular, we aim at computing the maximum likelihood:

$$\text{argmax}_{\pi,\mu,\Sigma} \ln(P(x|\pi, \mu, \Sigma))$$

The computed solution for this problem is:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n$$

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k)(x_n - \mu_k)^T$$

$$\pi_k = \frac{N_k}{N}, \text{ with } N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$

Now, to determine **z**

Expectation maximization exploits the formulas that we derived by *implementing them into an iterative two-step algorithm that is repeated until convergence.* These two steps are:

Expectation: Given the parameters of the distr., we compute the posterior of z.

Maximization: Given the prediction of the posterior, we compute the parameters of the mode.

The algorithm begins by initializing the parameters at step 0 with random values.

We start from the **expectation** step (E), where we compute the posterior given the parameters. The second step is **maximization** (M), where we do the opposite by taking the previously computed posterior and compute the new parameters.

The E and M steps are repeated until convergence. This method will output all the parameters.

The **Expectation-Maximization** (EM) algorithm is a pivotal method in unsupervised machine learning, particularly effective in scenarios **where the data involves hidden or latent variables**. Its primary function is to **find parameter estimates for statistical models**, where the model depends on unobserved latent variables. EM iteratively alternates between performing an expectation (E) step, which creates a function for the expectation of the log-likelihood evaluated using the current estimate for the parameters, and a maximization (M) step, which computes parameters maximizing the expected log-likelihood found on the E step.

14.4.1 Key Features of EM Algorithm

- **Convergence**: The algorithm converges to a local maximum likelihood, making it a reliable method for parameter estimation in complex scenarios.
- **Latent Variable Estimation**: It is particularly known for its ability to provide estimates of the latent variables, denoted as z_{nk}. This makes it suitable for scenarios where not all variables affecting the system are observable.
- **Extended K-means**: The EM algorithm can be viewed as a **probabilistic extension of the K-means clustering algorithm**. Unlike K-means, which hard assigns data points to clusters, EM assigns data points probabilistically.
- **Generalization to Other Distributions**: While often associated with Gaussian distributions, EM can be generalized to work with a broad range of distributions, enhancing its applicability.

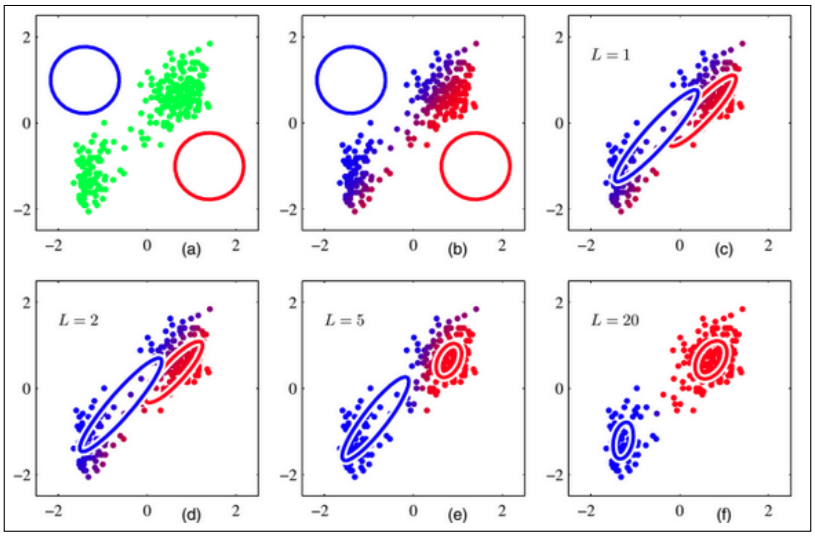


Figure 14.3

14.5 General EM Problem

The EM algorithm is applied under the following setup:

- **Observed Data** $X = \{x_1, \ldots, x_N\}$: The dataset consisting of N observed data points.
- **Unobserved Latent Values** $Z = \{z_1, \ldots, z_N\}$: A set of latent variables corresponding to each data point in X, not directly observable.
- **Parametrized Probability Distribution** $P(Y|\theta)$: The model assumes a probability distribution over the observed and latent data, parameterized by θ. Here, $Y = \{y_1, \ldots, y_N\}$ represents the complete data, where each y_i is a function of corresponding x_i and z_i.
- **Parameters** θ: The set of parameters that the algorithm aims to optimize.

Objective: The goal is to find the parameter set θ^* that locally maximizes the expected value of the logarithm of the likelihood $E[\ln P(Y|\theta)]$.

Applications:

- **Unsupervised Clustering**: Grouping data into clusters without predefined labels.
- **Bayesian Networks**: For learning the structure and parameters of Bayesian networks.
- **Hidden Markov Models**: Used in the parameter estimation of HMMs, especially in sequence data analysis.

14.6 General EM Method

The EM algorithm involves two main steps:

1. **E step (Estimation)**: Calculate the likelihood function $Q(\theta'|\theta)$, which is defined on the complete data set $Y = X \cup Z$. This step uses the observed data X and the current estimate of the parameters θ to estimate the distribution over the complete data Y:

$$Q(\theta'|\theta) \leftarrow E[\ln P(Y|\theta')|\theta, X]$$

   This step involves estimating the expected value of the log-likelihood with respect to the conditional distribution of Z given X under the current estimate of the parameters.

2. **M step (Maximization)**: Update the parameter θ to θ' that maximizes the Q function obtained in the E step:

$$\theta \leftarrow \arg\max_{\theta'} Q(\theta'|\theta)$$

   In this step, the algorithm finds the parameter values that maximize the likelihood function estimated in the E step.

So basically, merging everything together:

- **Unsupervised Learning**: The EM algorithm is a cornerstone in unsupervised learning, useful for dealing with both observed and unobserved (latent) variables.
- **Clustering Applications**: It is particularly useful for clustering when labeled data are not available, allowing for a more nuanced grouping based on probabilistic models.
- **General Method for Likelihood Estimation**: The EM algorithm provides a general framework for estimating the likelihood in mixed distributions, applicable to both discrete and continuous latent variables.

14.7 Summary: Unsup. Learning

When dealing with unlabelled or partially labeled data, we have a **unsupervised learning** problem . **Gaussian Mixture Models** (GMM) theorize data as stemming from multiple Gaussian distributions, each with its own mean and covariance, combined in a weighted manner. By assuming the data originates from this mix, we generate new data by selecting a Gaussian based on predefined weights, then sampling from that Gaussian. This process enables us to reconstruct the parameters of these distributions from the data, assuming the number of Gaussians is known.

$$P(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

K-means clustering, is a method used to group data into clusters. The "K" in K-means refers to the number of clusters you want to create. Initialize by choosing k clusters, start with k single-sample clusters, then assign remaining samples to the nearest cluster, updating **centroids** . Reassign samples to nearest centroids, updating as needed, until cluster memberships stabilize. *This requires pre-set cluster number, it's sensitive to **initial setup**, risking local optima, it's not outlier-robust, with outliers potentially skewing centroids, clusters tend to be circular due to reliance on distance metrics.* We **drop the assumption** of covariance and prior prob being the same for every distrib. . **Expectation Maximization**, is an algorithm that aims to find the parameter set θ^* that locally max the *exp. value of the log of the likelihood* $E[\ln P(Y|\theta)]$.

1. **E step**: Computes the expected log-likelihood $Q(\theta'|\theta)$ using current estimates θ, for observed data X and unobserved data Z:

$$Q(\theta'|\theta) \leftarrow \mathbb{E}[\ln P(Y|\theta')|\theta, X]$$

2. **M step**: Updates θ to maximize $Q(\theta'|\theta)$:

$$\theta \leftarrow \arg\max_{\theta'} Q(\theta'|\theta)$$

This iterates until convergence.

Chapter 15

Dimensionality Reduction

In many cases (like images) we find ourselves to work with very high dimensional inputs. But often it happens **that not all the combination of the input are meaningful**. In fact actual data may present a much lower variability (*for example if we are analyzing a dataset of numbers, only certain combination of pixels make sense in the context of the dataset*) The field of data reduction studies the possible transformation that can be applied to the input space in order to obtain compact (but still meaningful) representation of it in lower dimensions. When performing these transformations the risk is to lose information, or obtaining only partial representations of the problem. The following methods aim at finding **the best tradeoff between dimensionality reduction and information retention**.

15.1 Principal Component Analisys (PCA)

Imagine that we want to apply a transformation of a 2D dataset in a single dimension. What we do is basically projecting every point in 2 dimensions onto a 1D line, What we want is the transformation that guarantees the minimum error (meaning the maximum retention of information).

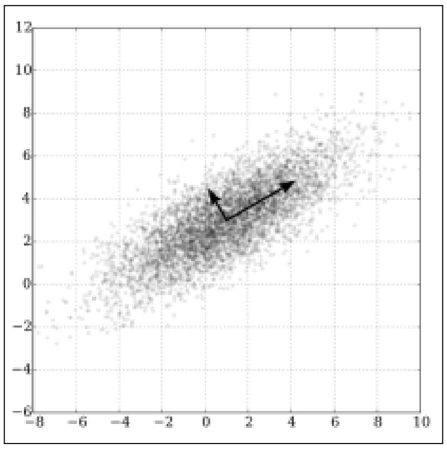


Figure 15.1. Dataset to reduce

In general, the best choice is the one that follows the direction of maximum variance of the data

15.1.1 PCA - Variance Maximization

Given data $\{x_n\} \in \mathbb{R}^d$ we aim to maximize data variance after projection to some direction $\mathbf{u}_1$ (unit vector)

The projected points are: $x_n^T \mathbf{u}_1$, **Note:** $\mathbf{u}_1^T \mathbf{u}_1 = 1$

First, we find the **mean of the data points** $\bar{x}$. Then we find the datacentered matrix, which is the result of rescaling all data in order to have mean 0:

$$X = \begin{bmatrix} (x_1 - \bar{x})^T \\ \vdots \\ (x_N - \bar{x})^T \end{bmatrix}$$

This procedure is also called **normalization** of the input data. We can now fix a certain direction with the goal of using it to find the direction of maximum variance.

Having **variance of projected points:** $\frac{1}{N} \sum_{n=1}^{N} \left[\mathbf{u}_1^T (\mathbf{x}_n - \mathbf{u}_1^T \bar{\mathbf{x}}) \right]^2 = \mathbf{u}_1^T S \mathbf{u}_1$

What we want is to solve the maximization problem by finding the direction that maximizes the variance: $\max_{\mathbf{u}_1} \mathbf{u}_1^T S \mathbf{u}_1$.

Solution

Setting the derivative with respect to $\mathbf{u}_1$ to zero, we have

$$S\mathbf{u}_1 = \lambda_1 \mathbf{u}_1$$

$\mathbf{u}_1$ must be an eigenvector of S by definition. Left-multiplying by $\mathbf{u}_1^T$ and using $\mathbf{u}_1^T \mathbf{u}_1 = 1$, we have

$$\mathbf{u}_1^T S \mathbf{u}_1 = \lambda_1$$

which is the variance after the projection.

Variance is maximal when $\mathbf{u}_1$ is the eigenvector corresponding to the largest eigenvalue λ_1. This is called the **first principal component** .

Given a dataset, we compute the data-centered matrix and the covariance matrix . From , we extract the eigenvalues and we order them in a decreasin order. If we are transforming the data into a D-dimensional space, we take the D highest eigenvalues, that correspond to the ones that will be the optimum for the variance maximizaton. The largest higenvalue of the correlation matrix is called principal component.

15.2 PCA - Error Minimization

15.2.1 Data Point Representation

Each data point x_n in the dataset can be represented as a linear combination of a set of basis vectors u_i:

$$x_n = \sum_{i=1}^{d} \alpha_{ni} u_i$$

The coefficients α_{ni} in this combination **are crucial as they determine how much each basis vector contributes to the data point**. This representation is foundational in transforming the data into a new coordinate system.

The **orthonormality property** of the basis vectors simplifies the computation of the coefficients. It allows us to express the coefficients α_{ni} as the dot products of x_n with each basis vector u_i:

$$x_n = \sum_{i=1}^{d} (x_n^T u_i) u_i$$

This reformulation is significant because it leverages the orthonormal nature of the basis vectors to simplify the representation of the data points.

The goal of PCA is to **reduce the dimensionality** of the data. To achieve this, we approximate x_n using a **lower-dimensional representation** $\tilde{x}_n$:

$$\tilde{x}_n = \sum_{i=1}^{m} z_{ni} u_i + \sum_{i=m+1}^{d} b_i u_i$$

This approximation involves **projecting the data points onto a subspace spanned by the first** m basis vectors. The remaining dimensions are accounted for by fixed coefficients b_i, which help in maintaining the structure of the data in the reduced dimensionality.

15.2.2 Approximation Error (MSE)

The quality of the approximation is quantified using the **Mean Squared Error (MSE):**

$$J = \frac{1}{N} \sum_{n=1}^{N} \|x_n - \tilde{x}_n\|^2$$

This metric calculates the average of the squared differences between the original data points x_n and their approximations $\tilde{x}_n$. A **lower** MSE indicates a **better** approximation of the original data in the reduced-dimensional space.

The **optimization process** involves adjusting the coefficients z_{ni} and b_i to minimize the approximation error. The coefficient z_{ni} represents the component of the data point in the subspace, while b_i is aligned with the mean of the data:

$$z_{ni} = x_n^T u_i, \quad i = 1, \ldots, m$$

$$b_i = \bar{x}^T u_i, \quad i = m+1, \ldots, d$$

This step is crucial in ensuring that the lower-dimensional representation retains as much information from the original data as possible.

15.2.3 Overall Approximation Error

The total approximation error is computed by considering the errors across all data points and all dimensions beyond the mth dimension:

$$J = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=m+1}^{d} (x_n^T u_i - \bar{x}^T u_i)^2 = \sum_{i=m+1}^{d} u_i^T S u_i$$

This comprehensive error measurement reflects how well the lower-dimensional space represents the variability of the original data.

15.2.4 Error Minimization with Constraint

Minimizing the error **involves a constraint** that ensures each basis vector u_i remains a unit vector:

$$\tilde{J} = \sum_{i=m+1}^{d} u_i^T S u_i + \lambda_i (1 - u_i^T u_i)$$

This constraint is crucial for maintaining the orthonormality of the basis vectors, which is a key assumption in PCA. The optimization leads to an eigenvalue equation, establishing that each u_i is an eigenvector of the covariance matrix S with a corresponding eigenvalue λ_i:

$$S u_i = \lambda_i u_i$$

This relationship is fundamental in PCA as it links the directions of maximum variance in the data (eigenvectors) to their magnitudes (eigenvalues).

15.2.5 Final Approximation Error

The final form of the approximation error is the **sum of the eigenvalues corresponding to the dimensions that were omitted**:

$$J = \sum_{i=m+1}^{d} \lambda_i$$

This sum provides a measure of the total variance lost in the dimensionality reduction process

In PCA, selecting the smallest eigenvalues for the error minimization process is equivalent to selecting the largest eigenvalues in a variance maximization formulation. This dual perspective highlights PCA's flexibility in either minimizing information loss or maximizing retained variance.

15.3 Probabilistic PCA

15.3.1 Linear Latent Variable Model

The probabilistic PCA leverages a probabilistic framework for PCA, where both the latent variables and their relationship with the observed data are modeled using Gaussian distributions. This approach provides a more flexible and probabilistic interpretation of PCA, allowing for the estimation of uncertainties and more nuanced data representations. In Probabilistic PCA, data points x are represented using lower-dimensional latent variables z. This approach assumes a linear relationship between the observed data and the latent variables:

$$x = Wz + \mu$$

Here, W is the transformation matrix mapping the latent variables z to the data space, and μ is the mean of the data points.

The latent variables z are **assumed to follow a Gaussian distribution**:

$$P(z) = \mathcal{N}(z; 0, I)$$

This assumption implies that the latent variables are centered around zero and have a standard normal distribution, indicating no prior bias in any specific direction in the latent space.

The relationship between the latent variables and the data is modeled as Linear-Gaussian:

$$P(x|z) = \mathcal{N}(x; Wz + \mu, \sigma^2 I)$$

This formulation captures the idea that the observed data x, given the latent variables z, follows a Gaussian distribution with mean $Wz + \mu$ and covariance $\sigma^2 I$, where σ^2 represents the variance of the data around the linear transformation.

15.3.2 Marginal Distribution

The marginal distribution of the observed data x is obtained by integrating over all possible values of the latent variables:

$$P(x) = \int P(x|z)P(z)dz = \mathcal{N}(x; \mu, C)$$

with

$$C = WW^T + \sigma^2 I$$

This represents the overall distribution of the data, capturing both the contribution from the latent space and the inherent data variability.

15.3.3 Posterior Distribution

The posterior distribution of the latent variables given the observed data x is:

$$P(z|x) = \mathcal{N}(z; M^{-1}W^T(x - \mu), \sigma^2 M)$$

with

$$M = W^T W + \sigma^2 I$$

This distribution provides insights into the latent variable values that are most likely to have generated the observed data, incorporating both the linear transformation and the variance in the data.

15.3.4 Maximum Likelihood PCA

Maximum Likelihood PCA provides a statistical basis for PCA by maximizing the likelihood of the observed data given the model parameters. This approach offers a rigorous way to estimate the parameters and is closely related to the probabilistic PCA model. Additionally, it can be solved using algorithms like the **Expectation-Maximization** (EM) algorithm for more complex scenarios where closed-form solutions may not be directly applicable.

Maximum likelihood PCA involves **finding the parameters that maximize the likelihood of the observed data** X. The objective is to find the parameters W, μ, and σ^2 that maximize the likelihood function:

$$\arg\max_{W,\mu,\sigma^2} \ln P(X|W,\mu,\sigma^2) = \sum_{n=1}^{N} \ln P(x_n|W,\mu,\sigma^2)$$

Here, X represents the observed data, W is the transformation matrix, μ is the mean of the data, and σ^2 represents the variance.

15.3.5 Closed-Form Solution

By setting the derivatives of the likelihood function to zero, we can find closed-form solutions for the parameters:

1. **Mean**:

$$\mu_{ML} = \bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n$$

The maximum likelihood estimate of the mean μ_{ML} is simply the average of the observed data points.

2. **Transformation Matrix (W)**:

$$W_{ML} = \ldots$$

The maximum likelihood estimate of the transformation matrix W_{ML} is more complex and depends on the eigenvalues and eigenvectors of the covariance matrix S. This relationship is not trivial and involves understanding how the directions of maximum variance (captured by the eigenvectors) contribute to the data structure.

3. **Variance**:

$$\sigma^2_{ML} = \ldots$$

The variance σ^2_{ML} is estimated as part of the maximum likelihood solution, reflecting the variability of the data around the model.

W depends on the eigenvalues and eigenvectors of S

This statement **underscores the importance of the covariance matrix** S in determining the transformation matrix W. The eigenvalues and eigenvectors of S play a crucial role in identifying the principal components in PCA.

15.4 Non-Linear Latent Variable Models

Traditional approaches like PCA assume linear relationships between data and latent variables. However, real-world data often display non-linear patterns, necessitating the use of non-linear models for more effective analysis and transformation.

Neural networks are adept at modeling complex, non-linear relationships, making them suitable for scenarios where linear models are insufficient. Their flexibility and computational capability enable them to handle intricate data transformations in non-linear latent variable modeling.

15.4.1 Autoencoders

Autoencoders are **neural networks** characterized by having input and output layers of the same dimension, with a smaller hidden layer (**the bottleneck**). This architecture allows them to **compress the input into a lower-dimensional space** and then reconstruct it back to the original dimension, effectively **learning a dense representation of the data.**

Designed specifically for high-dimensional data such as images, **convolutional autoencoders** utilize **convolutional layers** to efficiently encode and decode data. They are particularly effective at capturing **spatial hierarchies** and patterns in image data. Autoencoders can be effectively utilized for **anomaly detection** by training them on 'normal' data. They learn to represent typical patterns, and deviations in reconstruction from this learned representation can indicate anomalies.

15.4.2 Generative Models

Generative models like Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) offer advanced capabilities in data generation and manipulation.

Variational Autoencoders (VAEs)

VAEs introduce a probabilistic approach to autoencoders. Instead of encoding inputs to a specific point in latent space, **they encode them to distributions**. This allows for more controlled and **interpretable** generation of new data samples.

Generative Adversarial Networks (GANs)

GANs consist of: a generator and a discriminator. These networks are trained simultaneously in a competitive manner, where the generator aims to produce realistic data, and the other tries to distinguish between real and generated data.

15.5 Dimensionality Reduction: Summary

In high-dimensional data like images, not all input combinations are meaningful. Data reduction methods aim to transform input space for compact, yet meaningful lower-dimensional representation, balancing dimensionality reduction with information retention.

Principal Component Analisys is a transformation of a dataset along its **principal component** (eigenvector corresponding to the maximum eigenvalue), that is the direction of maximum variance.
For the **error minimization** we can, first, represent each data point as a linear combination of a set of basis vectors, we can choose the eigenvectors as a basis $x_n = \sum_{i=1}^{d}(x_n^T u_i)u_i$ We can approximate x_n with a **lower-dimensional representation of** $\bar{x}_n$, $\tilde{x}_n = \sum_{i=1}^{m} z_{ni}u_i + \sum_{i=m+1}^{d} b_i u_i$, quality of approximation is quantified using the **mean squared error**, the **lower**, the **better**.

$$J = \frac{1}{N}\sum_{n=1}^{N} \|x_n - \tilde{x}_n\|^2$$

The **optimization process** involves adjusting the coefficients z_{ni} and b_i to minimize the approximation error. The coefficient z_{ni} represents the component of the data point in the subspace, while b_i is aligned with the mean of the data:

$$z_{ni} = x_n^T u_i, \quad i = 1, \dots, m$$

$$b_i = \bar{x}^T u_i, \quad i = m+1, \dots, d$$

We can calculate the Overall Approximation Error (computing considering all error), we can also impose a constraint that ensures each basis vector remains a unit vector, but the **final approximation error** is obtained by the sum of the eigenvalues corresponding to the omitted dimensions.
Probabilistic PCA assumes a linear representation between data and latent variables $x = Wz + \mu$, latent variables assumed to follow a **Gaussian distribution**.
Maximum Likelihood PCA involves **finding the parameters that maximize the likelihood of the observed data** X.

$$\arg\max_{W,\mu,\sigma^2} \ln P(X|W,\mu,\sigma^2) = \sum_{n=1}^{N} \ln P(x_n|W,\mu,\sigma^2)$$

Here, X represents the observed data, W is the transformation matrix, μ is the mean of the data, and σ^2 represents the variance. By setting the derivatives of the likelihood function to zero, we can find closed-form solutions for the parameters. **Non-linear Latent Variable models** are often more useful when talking about real-life data. One example of this are **Autoencoders**, same dimensionality input-output but with a bottleneck at the center of the Neural Networks.

Chapter 16

Reinforcement Learning and MDPs

16.1 Dynamic Systems

The classical view of a dynamic system looks like this:

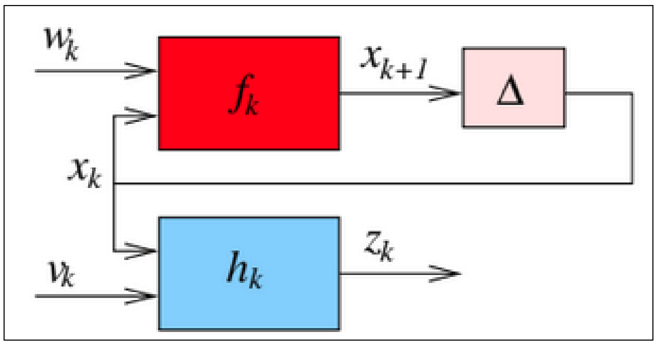


Figure 16.1. Dynamic System Representation

Where

- x: **state**
- z: **observations**
- w, v: **noise**
- f: **state-transition model**
- h: **observation model**

The state is usually the **configuration** of the system, it's possible that the system is only **partially observable**. For every system we will have a **set of states**. a **set of Actions**, a **set of observations**, a **transition function** and a **reward function**.

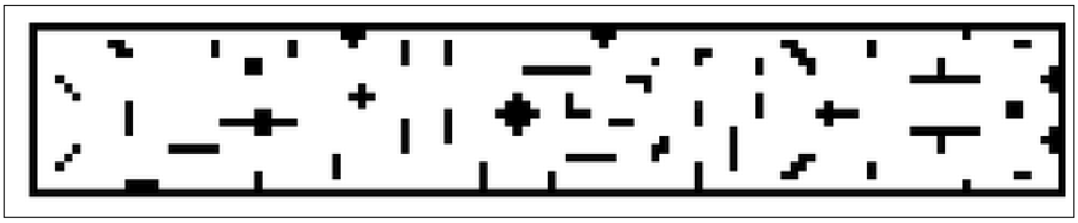


Figure 16.2. Deterministic example

GOAL: Reaching the right-most side of the environment from any initial state.

MDP: $\langle X, A, \delta, r\rangle$, *(States-Actions-Transition-Rewards)*

- $X = \{(r, c) \mid \text{coordinates in the grid}\}$
- $A = \{\text{Left}, \text{Right}, \text{Up}, \text{Down}\}$
- δ: cardinal movements with no effects (i.e., the agent remains in the current state) if the destination state is a black square
- r: 1000 for reaching the right-most column, -10 for hitting any obstacle, 0 otherwise

16.2 Markov Decision Processes (MDPs)

Markov Decision Processes are an abstraction for learning through goal-directed interaction in reinforcement learning. They model the interaction where an agent's actions influence not only immediate rewards but also probabilistically affect the future state of the environment.

- Interactions occur in discrete time steps: state, action, reward, new state.
- The objective is to maximize a function of the reward, often within sequences called *episodes*.

16.2.1 Markov Property

The Markov property is fundamental in MDPs, stating that the future state of a system depends solely on the current state and action, irrespective of past history.

- Simplifies problem-solving by focusing on current state and action.
- Enables use of state value or action value concepts without considering the entire history of transitions.

Given an MDP, we want to find an optimal policy, which is a function $\pi : X \to A$ which for each state $x \in X, \pi(x) \in A$ is the optimal action to be executed in such state, the way we measure **optimality** is maximizing the cumulative reward.

16.2.2 Partially Observable Environments

In many real-world scenarios, agents do not have complete observability of the environment, leading to Partially Observable Markov Decision Processes (POMDPs).

- **Stateless Deterministic Policies**: Treat observations as environmental states, but may not always provide optimal behavior.
- **Stateless Stochastic Policies**: Introduce randomness to avoid ambiguous situations, but finding an optimal policy is NP-Hard.
- **Policies with Internal State**: Use memory of past actions and observations. Techniques include recurrent neural networks and Hidden Markov Models for learning environment models and transitions.

In summary, MDPs are key in reinforcement learning, characterized by the Markov property, and are essential for modeling agent-environment interactions. POMDPs extend MDPs to scenarios with incomplete observability, employing various policy strategies to manage uncertainties in state perception.

16.2.3 Optimal Policy in Markov Decision Processes (MDPs)

Optimality in an MDP is about maximizing the expected cumulative discounted reward. The value function under a policy π for a state x_1 is defined as:

$$V_\pi(x_1) \equiv \mathbb{E}[\bar{r}_1 + \gamma \bar{r}_2 + \gamma^2 \bar{r}_3 + \ldots]$$

where $\bar{r}_t = r(x_t, a_t, x_{t+1})$, $a_t = \pi(x_t)$, and $\gamma \in [0, 1]$ is the discount factor for future rewards.

The value function differs based on whether the environment is deterministic or non-deterministic/stochastic.

- **Deterministic Case**: $V_\pi(x) \equiv r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots$
- **Non-deterministic/Stochastic Case**: $V_\pi(x) \equiv \mathbb{E}[r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots]$

The optimal policy π^* is defined as:

$$\pi^* \equiv \arg\max_\pi V_\pi(x), \forall x \in X$$

A policy π^* is optimal if and only if for any other policy π, the value function under π^* is greater than or equal to that under π for all states x:

$$V_{\pi^*}(x) \geq V_\pi(x), \forall x$$

For infinite horizon problems, a stationary MDP always has an optimal stationary policy. This ensures that there is always a best strategy for decision-making that remains consistent over time.

16.3 One-State MDPs

This is basically the simpler version of studying a MDP, every MDP could be considered a moltitude of these. We can distinguish multiple scenarios based on the $r(a_i)$ function.

Is Deterministic?	**Is Known?**	**Optimal Policy**	**Comput. Cost**
Yes	Yes	$\pi^*(x_0) = \mathbf{argmax}_{a_i} r(a_i)$	1 (Constant)
Yes	No	Try every action for the state and find every reward, then $\pi^*(x_0) = \mathbf{argmax}_{a_i} r(a_i)$	$\|A\|$
No	Yes	$\pi^*(x_0) = \mathbf{argmax}_{a_i} E[r(a_i)]$	$\approx$ *Constant?*
No	No	• Create a data structure • Choose an action, obtain reward • Update data structure • Repeat until **termination condition** • Use the structure to determine best action	$T >> \|A\|$

Table 16.1. One-State MDPs if $r(a_i)$ is known or deterministic

16.4 Exploitation-Exploration

Balancing *exploitation* and *exploration* is one of the main challenges of Reinforcement Learning algorithms. The concept essentially involves choosing the relative frequency with which to select, at each time step, **the best possible action** according to our estimates (an action called **greedy**) or an action called **explorative**.

Choosing a **greedy action** is obviously very effective based on good estimates but is a poor strategy if the estimates of the environment are not particularly good. Choosing an **explorative** action, on the other hand, means deliberately choosing a suboptimal action. The reason for this choice relates to obtaining new information about the environment since the "best action" is always selected based on our current estimates, regarding the states that have been visited and the actions that have been taken, but this risks excluding the visit of states that, even if temporarily seem suboptimal, over the long term can lead the agent to perform better.

In practice, by obtaining more information about the environment, our estimates tend to improve, and the next greedy actions are *better informed* and acquire more value. However, this effect needs to be balanced because by always choosing an explorative action, you end up "always" taking a suboptimal action and never converging to an optimal or near-optimal policy.

There are different approaches to this problem, which do not have a single objective solution for all cases. The two approaches that will be implemented in the following analysis are:

- **ϵ-greedy Policy**, where ϵ is the probability at each step (an individual interaction with the environment) of choosing an **explorative** action (and thus 1-ϵ is the probability of choosing a **greedy** action). This type of policy will be the main type used in the implementations of the algorithms discussed in the next chapter, as it is often particularly easy to implement and with the right tuning of the ϵ parameter, also particularly effective. It is worth noting that in **stationary environments**, the most efficient theoretical result involves starting learning with a percentage of exploratory actions and gradually decreasing it with the increase in steps, allowing for the best of both worlds: high initial exploration (more states are analyzed and more information about the environment is obtained) and low final exploration (tending to converge to a result without being stuck in suboptimal policies). The explorative action is, in the simplest case, randomly chosen from the available ones, which is very simple to implement but means that this *action selection* policy has received many criticisms regarding its characteristic of having the same probabilities, in the selection of an exploratory action, both of selecting an action that results in a better long-term action path and of selecting a poor or suboptimal action even for the long term.
- **Soft-max Strategy**, where actions with higher Q values are assigned higher probabilities, but every action is assigned a non-zero probability.

  $$P(a_i|x) = \frac{e^{kQ(x,a_i)}}{\sum_j e^{kQ(x,a_j)}}$$

  Here, $k > 0$ determines how strongly the selection favors actions with high Q values. k may increase over time (first exploration, then exploitation).

16.5 Learning Approaches

Learning with Markov Decision Processes (MDPs) involves an agent interacting with an environment characterized by states X, actions A, transition probabilities δ, and rewards r. The primary goal is to determine the optimal policy π^* that guides the agent to take actions maximizing cumulative rewards. Unlike supervised learning, there are no explicit input-output pairs for training, as the agent learns by interacting with the environment.

In the context of the **k-Armed Bandit problem** , the agent faces a simplified MDP with a single state and k possible actions. Each action a_i yields a stochastic reward according to a Gaussian distribution with mean μ_i and standard deviation σ_i. To balance between exploration (trying new actions) and exploitation (choosing actions with the highest expected rewards), strategies like the ϵ-greedy approach are employed.

The training process involves updating action values based on observed rewards. For instance, the training rule $Q_n(a_i) \leftarrow Q_{n-1}(a_i) + \alpha[\bar{r} - Q_{n-1}(a_i)]$ iteratively refines the estimated value of each action a_i, where α is the learning rate and $\bar{r}$ is the average reward obtained from selecting a_i.

But, in many cases we have more than ONE STATE, we have MANY STATES, there are different approaches to this problem

16.5.1 Value Iteration

Temporal Difference Learning

First, let's introduce the simplest methods of Temporal Difference, TD(0). This class of methods (a subclass of the broader TD(λ)) contains those methods that learn from a single step, weighing only that step in the estimation update phase.

For a Temporal Difference method, we define the so-called **Temporal Difference Error**:

$$\delta_t = R_{t+1} + \gamma V(x_{t+1}) - V(x_t)$$

Where

- $V(x_t)$ is the estimate of the state value at time t.
- R_{t+1} is the reward at time $t+1$ (next state).
- γ is the discount factor, a coefficient that weighs how much the estimate of the *next* state should be considered in the update of the current estimate.

This Temporal Difference error is what updates the estimates in a TD(0) algorithm, looking one step into the future and changing the evaluation of the current state based on that of the next state reached. The *how* this next state is reached will be the main difference between the TD(0) algorithms that we will see later.

A TD(0) method that estimates the value of a state updates the estimate at each step according to the rule:

$$V(x_t) = V(x_t) + \alpha[\delta_t]$$

Or

$$V(x_t) = V(x_t) + \alpha[R_{t+1} + \gamma V(x_{t+1}) - V(x_t)]$$

Where

- α is the learning rate or step-size parameter, which determines how much to weigh the modification of the current estimate.

The TD(0) methods are a special case of the TD(λ) methods, which depend on a parameter λ, ranging from 0 to 1, in the case where, of course, such parameter is null. TD(λ) methods introduce an additional weight to apply to the TD error (δ_t) for each single visited state, this additional parameter is called **eligibility** of a state, and it is defined as:

$$e(x_t) = \textstyle\sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{x_t, s_k}$$

Where

- $\delta_{s,s_k} = \begin{cases} 1 & \text{if } \; s = s_k \\ 0 & \text{otherwise} \end{cases}$

And represents the **visit degree** of a particular state in the recent past. Consequently, regarding the parameter λ :

- If the parameter is 0, we have a TD(0) method.
- If the parameter is 1, it is approximately equivalent to updating all states based on the number of times they have been visited by the end of the episode (more similar to a Monte Carlo method).

Learning rate and convergence of TD(0) methods
Sometimes it is convenient to vary the learning rate from time to time. We indicate with $\alpha_n(a)$ the learning rate used to process the reward received after the n-th selection of action a, the choice $\alpha_n(a) = 1/\text{n}$ leads to the so-called *sample average method*, which is guaranteed to converge to the true values of the actions according to the law of large numbers. However, of course, convergence is not guaranteed for all choices of the sequence $\alpha_n(a)$. A well-known result in stochastic approximation theory provides us with the necessary conditions to ensure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

The first condition is necessary to ensure that the steps are large enough to overcome any initial conditions or random fluctuations. The second condition ensures that the steps eventually become small enough to guarantee convergence.
It can also be proven that for any fixed policy π, a TD(0) method converges to v_π. Convergence occurs *on average*, for a constant and sufficiently small learning rate, and with probability 1 if the learning rate is variable but decreases according to the stochastic approximation conditions stated before.

Q-Learning

Q-Learning is a Temporal-Difference Algorithm (a TD(0) algorithm to be precise) that aims to learn a Q function $Q(x, a)$, associated with a couple state-action.

This function can be defined as:

$$Q(x, a) = r(x, a) + \gamma \max_{a'} Q(\delta(x, a), a')$$

where γ is the discount factor, $r(x, a)$ is the immediate reward, and $\delta(x, a)$ is the resulting state after taking action a in state x.

To learn the Q-function, the agent iteratively updates its approximation $Q^\dagger$ using a training rule. The training rule involves updating $Q^\dagger(x, a)$ based on the immediate reward and the maximum expected return for the successor state x' and all possible actions a'.

The Q-learning algorithm for deterministic MDPs follows these steps:

1. Initialize the Q-function table entries.

2. Observation of the current state x.
3. Iteration over time t until termination:
   - Action selection.
   - Execution of the action.
   - Observation of the new state x' and collection of immediate reward.
   - Update of the Q-function table entry for $Q^{\dagger}(x, a)$ using the training rule.
     - $Q(x, a) = r(x, a) + \gamma \max_{a'} Q(x', a')$
4. Determination of the optimal policy $\pi^*(x)$ based on the learned Q-function.

This learning process converges to the optimal Q-function as the agent visits all state-action pairs infinitely often, ensuring that the learned Q-values converge to their true values.

Q-Learning (λ)

One step time difference:

$$Q^{(1)}(x_t, a_t) \equiv r_t + \gamma \max_a Q^{\wedge}(x_{t+1}, a)$$

Two steps time difference:

$$Q^{(2)}(x_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a Q^{\wedge}(x_{t+2}, a)$$

n steps time difference:

$$Q^{(n)}(x_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a Q^{\wedge}(x_{t+n}, a)$$

Blend all of these ($0 \leq \lambda \leq 1$):

$$Q_\lambda(x_t, a_t) \equiv (1-\lambda)\langle Q^{(1)}(x_t, a_t)\rangle + \lambda Q^{(2)}(x_t, a_t) + \lambda^2 Q^{(3)}(x_t, a_t) + \ldots$$

SARSA

SARSA is based on the tuple $\langle s, a, r, s_0, a_0\rangle$ ($\langle x, a, r, x_0, a_0\rangle$ in our notation).

$$Q_n^{\wedge}(x, a) \leftarrow Q_{n-1}^{\wedge}(x, a) + \alpha \left[r + \gamma Q_{n-1}^{\wedge}(x_0, a_0) - Q_{n-1}^{\wedge}(x, a)\right]$$

a_0 is chosen according to a policy based on the current estimate of Q. SARSA is an on-policy method; it evaluates the current policy.

16.5.2 Policy Iteration

Use directly π instead of $V(x)$ or $Q(x, a)$.
Parametric representation of π: $\pi_\theta(x) = \max_{a \in A} Q_\theta^{\wedge}(x, a)$.
Policy value: $\rho(\theta)$ = expected value of executing π_θ.
Policy gradient: $\Delta_\theta \rho(\theta)$.

Policy Gradient Algorithm

Policy gradient algorithm for a parametric representation of the policy $\pi_\theta(x)$

```
**choose** $\theta$
**while** termination condition **do**
    estimate $\Delta_\theta \rho(\theta)$ (through experiments)
    $\theta \leftarrow \theta + \eta \Delta_\theta \rho(\theta)$
**end while**
```

General method

```
$\pi \leftarrow$ InitialPolicy
**while** termination condition **do**
    compute $\{R_1, ..., R_t\}$, random perturbations of $\pi$
    evaluate $\{R_1, ..., R_t\}$
    $\pi \leftarrow$ getBestCombinationOf($\{R_1, ..., R_t\}$)
**end while**
```

Note: in the last step we can simply set $\pi \leftarrow \arg\max_{R_j} F(R_j)$ (i.e., hill climbing).

Perturbations are generated from π by $R_i = \{\theta_1 + \delta_1, ..., \theta_N + \delta_N\}$

with δ_j randomly chosen in $\{-j, 0, +j\}$, and j is a small fixed value relative to θ_j.

Combination of $\{R_1, ..., R_t\}$ is obtained by computing for each parameter j:

- Avg-,j: average score of all R_i with a negative perturbations
- Avg0,j: average score of all R_i with a zero perturbation
- Avg+,j: average score of all R_i with a positive perturbations

Then define a vector $A = \{A_1, ..., A_N\}$ as follows

$$A_j = \begin{cases} 0 & \text{if } \mathrm{Avg}_{0,j} > \mathrm{Avg}_{-,j} \text{ and } \mathrm{Avg}_{0,j} > \mathrm{Avg}_{+,j} \\ \mathrm{Avg}_{+,j} - \mathrm{Avg}_{-,j} & \text{otherwise} \end{cases}$$

and finally $\pi \leftarrow \pi + \frac{A}{|A|}\eta$

16.6 Hidden Markov Models and Partially Observable MDPs

16.6.1 Hidden Markov Models

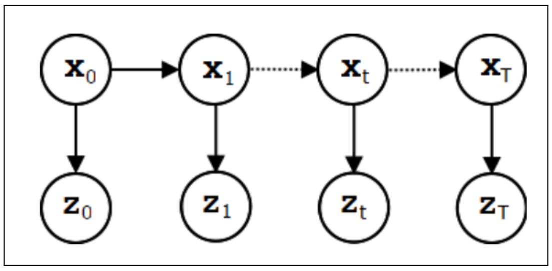


Figure 16.3. Hidden Markov Models

HMM Factorization

Application of chain rule on HMM:

$$P(x_{0:T}, z_{1:T}) = P(x_0)P(z_0|x_0)P(x_1|x_0)P(z_1|x_1)P(x_2|x_1)P(z_2|x_2)\dots$$

HMM Inference

Given HMM $= h(X, Z, \pi_0)$,

Filtering

Filtering refers to estimating the current state of the system given all observations up to the current time step. It is typically computed using the forward algorithm and provides the posterior distribution of the current state given the observations

$$P(x_T = k|z_{1:T}) = \alpha_k P_T \sum_j \alpha_j^T$$

Smoothing

Smoothing refers to estimating the past states of the system given all observations up to the current time step. It is typically computed using both forward and backward algorithms and provides the posterior distribution of past states given all observations

$$P(x_t = k|z_{1:T}) = \alpha_k^t \beta_k^t P_j \alpha_j^t \beta_j^t$$

16.6.2 Forward Step

This step computes the forward probabilities α_k^t, which represent the probability of being in state k at time t, given all observations up to time t.

1. **Initialization:** At time $t = 0$, the forward probability for each state k is calculated based on the initial state distribution π_0 and the emission probabilities $b_k(z_0)$, where z_0 is the first observation.
2. **Iteration:** For each subsequent time step t, the forward probabilities are updated based on the previous time step's probabilities and the transition probabilities A_{jk} between states j and k, weighted by the emission probabilities $b_k(z_t)$ for the current observation z_t.

16.6.3 Backward Step

This step computes the backward probabilities β_k^t, which represent the probability of observing the sequence of future observations given that the system is in state k at time t.

1. **Initialization:** At the last time step T, the backward probabilities for each state k are initialized to 1, indicating certainty about future observations.
2. **Iteration:** For each time step t from $T-1$ down to 1, the backward probabilities are updated by summing over the probabilities of transitioning from state k to state j, weighted by the emission probabilities $b_j(z_{t+1})$ for the next observation z_{t+1}, and the backward probability β_j^{t+1} for the next time step $t + 1$.

These forward and backward steps are crucial components of the forward-backward algorithm, which is used for various tasks in HMMs, such as smoothing, filtering, and parameter estimation. By combining forward and backward probabilities, one can efficiently estimate the probability distributions over hidden states at each time step and over entire sequences of observations.

16.6.4 POMDP

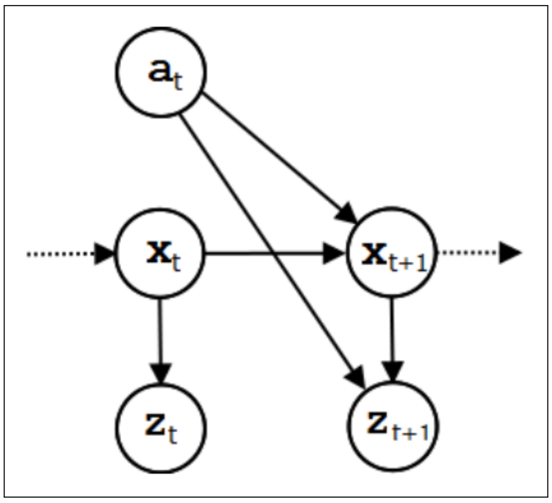


Figure 16.4. Partially Observable MDP

$$\text{POMDP} = \langle hX, A, Z, \delta, r, o \rangle$$

Where:

- X is a set of states
- A is a set of actions
- Z is a set of observations
- $P(x_0)$ is a probability distribution of the initial state
- $\delta(x, a, x_0) = P(x_0|x, a)$ is a probability distribution over transitions
- $r(x, a)$ is a reward function
- $o(x_0, a, z_0) = P(z_0|x_0, a)$ is a probability distribution over observations

To solve a POMDP we describe it as a MDP in the **belief states**. A belief state refers to the **probability distribution over the set of possible states** of the system. Unlike in a standard MDP where the current state is fully observable, in partially observable environments, the agent doesn't have direct access to the true state but instead receives observations that are **probabilistically related to the underlying states**. In such scenarios, the belief state captures the agent's uncertainty about which state the system is currently in.

16.7 Computing Belief States

Given current belief state b, action a, and observation z_0 observed after execution of a, compute the next belief state $b_0(x_0)$:

$$b_0(x_0) \equiv SE(b, a, z_0) \equiv P(x_0|b, a, z_0)$$

$$= P(z_0|x_0, b, a)P(x_0|b, a)$$

$$P(z_0|b, a) = P(z_0|x_0, a) \sum_{x \in X} P(x_0|b, a, x)b(x)$$

$$P(z_0|b, a) = o(x_0, a, z_0) \sum_{x \in X} \delta(x, a, x_0)b(x)$$

Belief MDP Transition and Reward Functions

Transition function

$$\tau(b, a, b_0) = P(b_0|b, a) = \sum_{z \in Z} P(b_0|b, a, z)P(z|b, a)$$

$$P(b_0|b, a, z) = \begin{cases} 1 & \text{if } b_0 = SE(a, b, z) \\ 0 & \text{otherwise} \end{cases}$$

Reward function

$$\rho(b, a) = \sum_{x \in X} b(x)r(x, a)$$

Value function in POMDP

$$V(b) = \max_{a \in A} \left[\rho(b, a) + \gamma \sum_{b_0} P(b_0|b, a)V(b_0) \right]$$

Replacing $\tau(b, a, b_0)$ and $\rho(b, a)$ and considering that $P(b_0|b, a, z) = 1$ if $b_0 = SE(a, b, z) = b_z^a$, and 0 otherwise:

$$V(b) = \max_{a \in A} \left[\sum_{x \in X} b(x)r(x, a) + \gamma \sum_{z \in Z} P(z|b, a)V(b_z^a) \right]$$

Value iteration for belief MDP

Discretize the distributions $b(x)$, apply value iteration on the discretized belief MDP.

A similar method can be devised for any MDP solving technique.

16.8 Reinforcement Learning: Summary

Reinforcement Learning Learning from interaction to achieve goals. Updates value estimates based on temporal differences. Balances exploration and exploitation with soft-max strategy, **Markov Decision Processes (MDPs)** is a Model for sequential decision-making. Defines states, actions, transitions, and rewards. Aims to find the optimal policy maximizing expected rewards.

- **State Transition Probability**: $P(s' \mid s, a)$
- **Expected Reward**: $R(s, a)$
- **Value Function**: $V^{\pi}(s) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right]$

Temporal Difference , class of algorithms, **Q-learning** , Off-policy learning method to learn the value of an action in a particular state. Updates Q-values using the maximum expected future rewards.

- **Temporal Difference Error**: $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
- **Value Update Rule**: $V(S_t) \leftarrow V(S_t) + \alpha\delta_t$
- **Q-value Update Rule**: $Q(s, a) \leftarrow Q(s, a) + \alpha\left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)$

Hidden Markov Models (HMMs) Models systems with hidden states emitting observable symbols. Estimates state probabilities given observed sequences. Uses forward-backward algorithms for filtering and smoothing.

- **Forward Algorithm**: $\alpha_t(i) = P(O_1, O_2, ..., O_t, Q_t = S_i \mid \lambda)$
- **Backward Algorithm**: $\beta_t(i) = P(O_{t+1}, O_{t+2}, ..., O_T \mid Q_t = S_i, \lambda)$

Partially Observable Markov Decision Processes (POMDPs) Merges MDPs with HMMs for uncertain environments. Maintains belief states to represent uncertainty. Maximizes value under partial observability for decision-making.

- **Belief State Update**:

$$b'(s') = \frac{1}{P(o' \mid a, b)} \cdot \sum_{s \in S} P(s' \mid s, a) \cdot P(o' \mid s') \cdot b(s)$$

- **Value of Belief State**:

$$V^{\pi}(b) = \sum_{a \in A} \pi(a \mid b) \cdot \left(R(b, a) + \gamma \sum_{b' \in B} P(b' \mid b, a) \cdot V^{\pi}(b')\right)$$