AIML: Artificial Intelligence and Machine Learning

Lecture Notes v1.12

Dr. Max A. Little

January 11, 2024

UNIVERSITY OF
BIRMINGHAM

Contents

Part I.

Introduction

Section 1.

Module overview

Relevant references for this section are **R&N** Section 1.1, **MLSP** Preface, and **PRLM**, Section 1 (up to start of Section 1.1).

1.1. Reference material

The main AI reference textbooks are (shorthand citation used in this module in **bold**):

- ▷ Russell and Norvig, 2010, *Artificial Intelligence: A Modern Approach*, 3rd Edition/4th Edition, Prentice Hall (**R&N**)
- ▷ Cormen, Leiserson, Rivest and Stein, 2022, *Introduction to Algorithms*, 4th Edition, The MIT Press (**CLRS**)

The main ML reference textbooks are:

- ▷ Bishop, 2006, *Pattern Recognition and Machine Learning*, Springer (**PRML**)
- ▷ Little, 2019, *Machine Learning for Signal Processing*, Oxford University Press (**MLSP**)
- ▷ Hastie and Tibshirani, 2008, *The Elements of Statistical Learning*, 2nd Edition, Springer (**H&T**)

Background mathematics textbooks:

- ▷ Gill, 2006, *Essential Mathematics for Political and Social Research*, Cambridge University Press (**Gill**)
- ▷ Stirzaker, 2003, *Elementary Probability*, Cambridge University Press (**Stirzaker**)
- ▷ Strang, 1991, *Calculus*, Wellesley-Cambridge Press (**Strang**)

Further reference textbooks:

- ▷ Kleinberg and Tardos, 2006, *Algorithm Design*, Pearson/Addison-Wesley
- ▷ Papadimitriou and Steiglitz, 1998, *Combinatorial Optimization*, Dover Books
- ▷ Murphy, 2012, *Machine Learning: A Probabilistic Perspective*, The MIT Press

1.2. How to use these lecture notes

These notes have been designed to accompany the lectures and tutorials. The **learning outcomes** for each section (that is, what you should know or be able to do after reading that section thoroughly) are listed at the end of each section.

In these notes, we have highlighted specific terminology which is used in AI and ML in **bold blue**. These are technical terms you will need to familiarize yourself with, because they are used to shortcut discussions of technical issues, so, coursework and exam questions may be phrased in these terms. Other terms which have been borrowed from other disciplines but are not central to the topic are emphasized with *italics*, and other incidental that are useful or required for the discussion are highlighted in **bold**.

1.3. Course content

This module introduces the core concepts of **artificial intelligence** (**AI**) and **machine learning** (**ML**). The principal focus of the module will be on the underlying principles such as **knowledge representation**, **optimization**, **probability**, and **statistical learning**. On successful completion of this module, the student should be able to:

- ▷ Explain and discuss a range of problems and techniques in AI and ML
- ▷ Compare AI and ML techniques, describing their strengths and limitations
- ▷ Apply AI and ML techniques to solve problems

1.4. Overview of AI and ML

AI and ML address challenging real-world problems which require the manipulation or processing (**computation**) with information (**data**) in some way, to come up with rational decisions given this information. In principle, all these problems could be solved manually, but humans, unless paying careful attention, easily make mistakes or get tired, so this approach is not practical for many modern, large-scale problems.

While all these problems could be solved by **exhaustively** generating or all possible solutions and testing them out one-by-one to select the best solution (AI), or storing all input-output patterns in memory and then just looking up the solution (ML), this is usually **infeasible**. A main motivation therefore of AI and ML is in coming up with "smart" solutions to solving complex problems under conditions of limited computational resources.

Examples of real-world problems which can be solved, even if approximately, using techniques from AI and ML include:

- ▷ GPS **navigation**; finding an optimal road route, possibly including waypoints and incorporating real-time traffic updates
- ▷ **Scheduling** limited uplink/downlink radio channels for deep space communication
- ▷ **Image processing**; such as colorizing black and white photographs
- ▷ Text-based language **translation**

Learning outcomes for section 1

- ▷ Demonstrate understanding of the practical value of AI and ML algorithms and where they are applicable.
- ▷ Be able to explain and reason broadly about the fundamental differences between symbolic AI and (statistical) ML.

Section 2.

Essential mathematical background

A good introduction to the necessary mathematics (ignore the examples), can be found in **Gill**, Section 1.2, Section 1.3, Section 1.4, Section 1.5, Section 1.6 and Section 1.7. These sections should be read now before going any further.

2.1. Essential mathematical concepts

To study the material in this course there are various background mathematical concepts and terminology that you need to know and understand. Some of these concepts are just a matter of understanding the notation, and others are derived results (theorems) which you should know. Other, more specialized mathematical concepts, notation and terminology will be introduced later in the lecture material where needed.

2.2. Mathematical notation in this module

In this course, we generally refer to natural numbers or integers using upper-case italics, i.e. $N \in \mathbb{Z}$ or $M \in \mathbb{N}$. A closed subset of the real is given by $[a, b]$, and an open set is $]a, b[$. A set of distinct values is written $\{a, b, c \ldots\}$. Names for special sets use calligraphic symbols e.g. $\mathcal{X}, \mathcal{Y}$. Real variables are written in lower-case italics, i.e. $x \in \mathbb{R}$. A sequence of N variables is written $x_1, x_2 \ldots, x_N$. Lower case letters generally refer to integer indices, as in x_n for $n = 1, 2, 3 \ldots, N$. A function is generally denoted $f(x)$ or $F(x)$. For differentiating once, we write $f'(x) = \frac{d}{dx} f(x)$; the partial derivative is written $\frac{\partial}{\partial x} f(x)$. Multidimensional integration over a set A is written $\int_A dx\, dy\, dz \ldots$ or just $\int_A dx$ if the context is clear. The probability of a set A is denoted $\Pr(A)$. A probability distribution function or density function is written $P(x)$ or $p(x)$.

2.3. Plotting functions

In order to get some idea about a function, it is often very helpful to visualize it. A good plot satisfies the following criteria:

1. It should be **informative** and clear:
   a) The **scale** of the axes should be adapted to the scale of the data, and/or the range of the function.
   b) Axes should be **labeled**.
   c) It should have a **legend** and/or a **caption** which explains what has been plotted and what any symbols and colours mean.

2. It should be **neat** and easy to read. Some tips for achieving this:
   a) Use **graph paper** and/or a **computer algebra** package (such as Mathematica).
   b) Draw straight lines using a **ruler**.
   c) Distinguish different lines and points using **colour** and/or different **line types** (i.e. solid, dashed, dotted etc.)

When plotting a function by hand, you should keep the following in mind:

1. Unless otherwise restricted, a function $f(x)$ on $\mathbb{R}$ maps each value of x to a unique point $y = f(x)$ on your graph, so, if there is any potential for ambiguity (for example, where a function has a jump) you should make the actual value clear using a thick point.
2. One way to plot a function qualitatively by hand, is to find a few important x_i values (e.g. where the function crosses zero, where there are singularities or jumps etc.), then connect the points $(x_i, f(x_i))$ as smoothly as possible. Where the points you have chosen just have convenience for your plotting, they should not be visible on the plot when it is finished.

Learning outcomes for section 2

▷ Knowledge of the elementary mathematical concepts used in this module.

▷ Familiarity with the mathematical notation used in this module.

▷ How to plot mathematical functions.

Part II.

Symbolic artificial intelligence

Section 3.

Combinatorial optimization

Relevant reference reading material for this section are **MLSP**, Section 1.8, Section 2.6; **CLRS**, Chapter 3 and **R&N**, Section A.1.

3.1. Optimizing objective functions

Many problems in **symbolic AI** can be formulated mathematically as **optimization problems**. These are formulated as **minimizing**[1] an **objective function** $F(X)$ with respect to a set of all possible **configurations**, $\mathcal{X}$:

$$X^\star = \arg\min_{X' \in \mathcal{X}} F\left(X'\right). \tag{3.1}$$

An **exact method/algorithm**, is guaranteed to find (one of) the **globally optimal** values of (3.1). Mathematically, we say that x is globally optimal if it satisfies,

$$F\left(X'\right) \geq F\left(X^\star\right), \ \forall X' \in \mathcal{X}. \tag{3.2}$$

By contrast, an **approximate method/algorithm** is only guaranteed to find a **locally optimal** value (a value which is "good enough"). Note that this locally optimal value could also be globally optimal, but the point is that *we cannot be sure of this*. Stated mathematically, the configuration x is locally optimal if, for some **subset** (some restricted part) $\mathcal{N} \subset \mathcal{X}$ of the whole set of possible configurations,

$$F\left(X'\right) \geq F\left(X^\star\right), \ \forall X' \in \mathcal{N}. \tag{3.3}$$

Exact methods are typically quite slow, whereas, approximate methods might be fast but are not necessarily reliable. To appreciate the difficulty of the optimization problem, we need to understand the kind of phenomena which can be encountered in objective functions (Figure 3.1). Unless the problem is **convex**, that is, it has only one minima which is also the global minima, then there can be multiple local minima and/or maxima. There can also be **saddle points** which are locally 'flat' but which are not maxima or minima. Minima/maxima can be relatively sharp or shallow.

3.2. Exhaustive search and combinatorial explosion

Any combinatorial optimization problem (3.1) can always be solved **exhaustively** (sometimes called **brute-force**), that is, by computing $F\left(X'\right)$ for all $X' \in \mathcal{X}$, and then just selecting an optimal one. While this is an exact method, for most real-world problems, this method is **intractable** due to **combinatorial explosion**. As an example, consider the classic computer science problem of **sorting**

[1] We could also *maximize* the same objective by replacing $F\left(x\right)$ with $-F\left(x\right)$, and the problem would be unchanged.

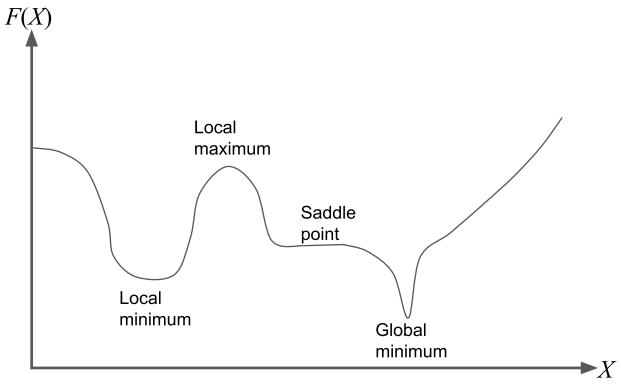

Figure 3.1.: Typical phenomena encountered in objective functions, $F(X)$ as the configuration X is varied over the set $\mathcal{X}$ of possible configurations.

the set of $N = 3$ numbers $\{5, 18, 2\}$ in ascending order; we can do this by testing every **permutation** (unique ordering) of the data and selecting one which is in the correct order.[2] The configuration set is,

$$\mathcal{X} = \{\{5, 18, 2\}, \{18, 5, 2\}, \{2, 5, 18\}, \{5, 2, 18\}, \{18, 2, 5\}, \{2, 18, 5\}\}$$

which we write as $|\mathcal{X}| = 6$ for the **problem size** $N = 3$, and we find that an optimal configuration is $X^\star = \{2, 5, 18\}$. It is easy to see that this is a way to guarantee finding a globally optimal solution. However, for any arbitrary problem size N, the number of permutations is given by the factorial, $|\mathcal{X}| = N!$ which grows extremely rapidly as N increases. For example, $10! = 3628800$ and $50! \approx 3.04 \times 10^{64}$ which is an astronomically large number of possible orderings and it would take even the faster supercomputer too long to test them all on any reasonable timescale. Thus, sorting by exhaustive search is generally impractical, and we need "smarter" algorithms. This is one of the main motivations for the discipline of AI.

3.3. Complexity classes and worst case algorithm performance

Problems, such as (3.1), are often usefully classified according to their **complexity class**. This is a mathematical way of broadly characterizing how rapidly a function value grows with its input, using the so-called **big-oh notation** (**asymptotic complexity**), $O()$. For computational problems such as (3.1), this big-oh function tells us the **worst case performance** of the algorithm, either in terms of how many steps it requires – **computational complexity** – or in terms of how much memory it requires – **space complexity**. See Table 3.1. Sorting by exhaustive permutation testing has computational complexity $O(N!)$.

Complexity classes provide a mathematical expression of the behaviour of all functions within its class, as $N \to \infty$ ("asymptotically"). So, the two functions $f(N) = N^2 + 3$ and $g(N) = N^2$ are in the same (polynomial) class, $O(N^2)$, even though g grows slower with N than f.

[2] We can express the objective function for this problem mathematically as $F(X) = 0$ if X is in ascending order, and $F(X) = 1$ otherwise.

Complexity class	Big-Oh notation
constant	$O(1)$
logarithmic	$O(\log N)$
linear	$O(Np)$
log-linear	$O(N \log N)$
polynomial	$O(N^p)$
exponential	$O\left(p^N\right)$
factorial	$O(N!)$

Table 3.1.: Big-oh notation for worst case complexity classes.

Generally speaking, algorithms which have polynomial computational or space complexity and better, are considered **tractable** (usable in practice), and problems which are exponential or worse, are considered **intractable** (not practically useful). The best algorithm has constant $O(1)$ complexity, which means that the number of computational steps it takes, does not in any way depend upon the size of the problem, whereas the worst algorithm has factorial $O(N!)$ complexity.[3] However, this does not mean that e.g. two polynomial complexity algorithms are always as good as each other: there is a substantial practical difference between $O(N^2)$ and $O(N^3)$, for instance. The main consideration for a desirable AI algorithm, is that it has the **best worst case complexity** as possible.

Most AI algorithms can be organized so as to **trade-off time for space** (and vice-versa): the number of steps to completion can be reduced by increasing the amount of memory required, and vice-versa. For instance, if the same (or similar) problem has to be solved repeatedly, then, if we have previously found the optimal solution x for some part $\mathcal{N}$ of the configuration set $\mathcal{X}$, we can store this optimal solution x in memory and then next time when we are asked to find the optimal solution in $\mathcal{N}$, we can solve it in $O(1)$ steps by simply retrieving the stored solution from memory. Of course, by doing this we have to use memory which otherwise would not be required. This trick is known as **memoization** and is widely used in practical AI algorithms, particularly **dynamic programming** which is discussed later.

Learning outcomes for section 3

- ▷ Express solving an AI problem through optimizing an appropriate mathematical objective with respect to a set of configurations.
- ▷ Define combinatorial explosion.
- ▷ Identify the complexity class of a function and the worst-case complexity class of an algorithm.
- ▷ Be able to formulate and apply an exhaustive solution to a combinatorial optimization problem.

[3] There are even worse complexity classes than factorial, but factorial complexity is always a "hard limit" in practice.

Section 4.

Exact algorithms

Relevant reference reading material for this section are **MLSP**, Section 2.6, and **CLRS**, Chapter 21.

4.1. Algorithm strategies

Many **algorithm strategies** for constructing practical exact methods have been identified in the discipline, for instance, **divide-and-conquer**, **branch-and-bound**, and **dynamic programming** and variants are widely used. Every strategy involves a distinct way of organizing information and computations such as to solve the problem exactly under the best worst case time and space complexity (or at least, the best known). It is important to understand that no single strategy appears best for all problems; the usefulness and/or computational efficiency of each strategy depends heavily upon the specific structure of the problem. The choice of algorithm strategy therefore requires understanding of the specifics of the problem itself and the computational resources available to solving it. Sometimes, the availability of other information about the problem can make a big difference to the choice of strategy and resulting algorithm efficiency.

4.2. Sequential decision processes (SDPs)

In this module, we will concentrate on **sequential decision processes (SDPs)** as they are conceptually easy to design and describe, efficient, and furthermore, encompass many well-known AI algorithms used in practice. An SDP algorithm is a **recursive** process which scans N input data items $x_1, x_2, \ldots, x_N$ in sequence, generating new configurations by **extending** the existing configurations using each input data item in turn, and **reducing** the set of candidate configurations by removing any which cannot be extended to an optimal configuration (Algorithm 4.1).

As a useful graphical aide to keep track of the calculations performed by this algorithm is a **computational configuration graph**. Different SDP strategies have recognizably distinct graphs. **Exhaustive (brute-force)** SDP algorithms have no means of applying reduction in step 3, therefore, the number of candidate configurations grows rapidly at each iteration, in a **full tree** structure (Figure 4.1).

At the other extreme, **greedy** SDP remove all but the **current best** candidate at each stage. As a result of this aggressive reduction, the graph is a tree with which never has more than k branches, where k is the worst case number of possible extensions at each stage (Figure 4.2).

Most SDPs lead to computation trees which lie somewhere in between the extremes of exhaustive and greedy, indeed. Note that, clearly, to retain algorithm exactness, the reduction step cannot remove any configuration which has the possibility of being extended to an optimal one in later iterations, so the computational efficiency of the algorithm depends heavily upon finding an effective reduction strategy.

Algorithm 4.1 Generic sequential decision process (SDP) for combinatorial optimization of AI problems.

- ▷ **Step 1**. *Initialization*: Start with $n = 0$, generate the root configuration(s) in the set of candidate configurations, S.
- ▷ **Step 2**. *Extension*: Set $n = n + 1$, and using data item x_n, extend all candidate configurations in S, and append these to S.
- ▷ **Step 3**. *Reduction*: Remove any candidate configurations from S which cannot be extended to an optimal configuration.
- ▷ **Step 4**. *Iteration*: If $n < N$, go back to step 2.
- ▷ **Step 5**. *Select best*: Select an optimal configuration $X^\star$ from the remaining candidate configurations in S.

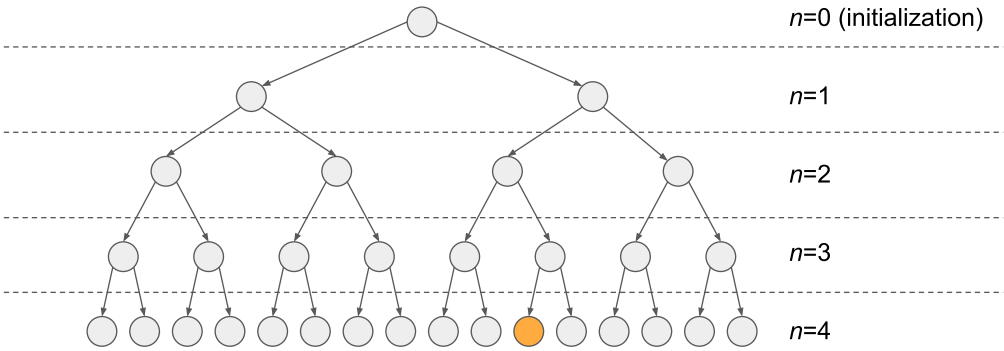

Figure 4.1.: Computational graph of a fully exhaustive sequential decision process (SDP) for solving combinatorial optimization problems in AI. Each circle is a configuration, at each stage n, each remaining candidate configuration at that stage is extended by the new data item x_n. After the final iteration (here, stage $n = 4$), an optimal configuration $X^\star$ (orange) is selected from the remaining candidates.

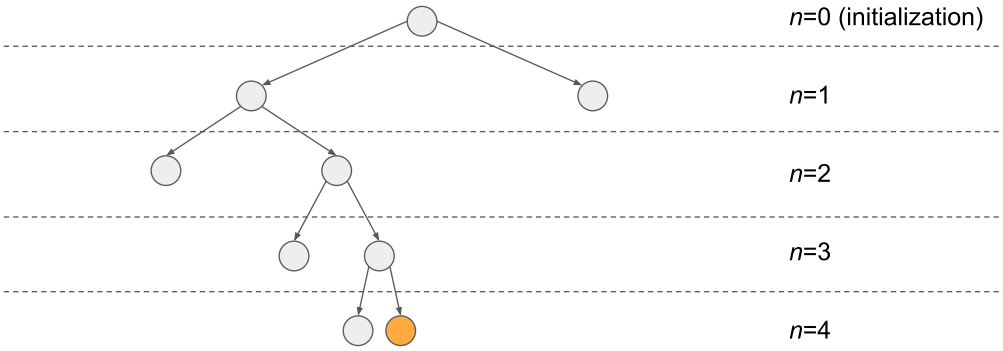

Figure 4.2.: Computational graph of a greedy sequential decision process (SDP) for solving combinatorial optimization problems in AI. Each circle is a configuration, at each stage n, each remaining candidate configuration at that stage is extended by the new data item x_n, and all but the next best candidate are removed from S. After the final iteration (here, stage $n = 4$), an optimal configuration $X^\star$ (orange) is selected from the remaining candidates.

A simple example of (greedy) SDP is **insertion sort** (Algorithm 4.2 and Figure 4.3). The greedy reduction step is valid here because clearly, an unsorted permutation cannot be extended to a sorted permutation by any sequence of insertions, thus unsorted permutations can never lead to sorted permutations, thus the algorithm is exact. At each stage $O(N)$ new permutations are generated, thus, since there are N stages, the entire algorithm has $O(N^2)$ time complexity. The algorithm is naive however, since there are much faster ways of constructing the permutations using a "smarter" data structure such as **binary search trees** which leads to a faster $O(N \log N)$ implementation (see CLRS, Section 12).

Learning outcomes for section 4

- ▷ Explain and apply the sequential decision process (SDP) algorithm for exactly solving combinatorial optimization problems in AI.
- ▷ Construct the computational graph for an SDP algorithm.
- ▷ Explain and apply exhaustive and greedy SDP algorithms.
- ▷ Determine the computational complexity of a given SDP algorithm.

Algorithm 4.2 A simple (naive) implementation of insertion sort which runs in $O\left(N^2\right)$ time.

▷ **Step 1**. *Initialization*: Start with $n = 0$, generate the root permutation $S = \{[\,]\}$.

▷ **Step 2**. *Extension*: Set $n = n + 1$, and using data item x_n, extend all candidate permutations in S by inserting x_n at every possible position in each candidate permutation, and append these new permutations to S, e.g. at stage $n = 3$, $[x_1, x_2]$ would be extended to $[x_3, x_1, x_2]$, $[x_1, x_3, x_2]$ and $[x_1, x_2, x_3]$.

▷ **Step 3**. *Reduction*: Remove all candidate permutations from S which are not in sorted order, if there are ties then select one arbitrarily.

▷ **Step 4**. *Iteration*: If $n < N$, go back to step 2.

▷ **Step 5**. *Select best*: Select a sorted configuration $X^\star$ from the remaining candidate permutations in S.

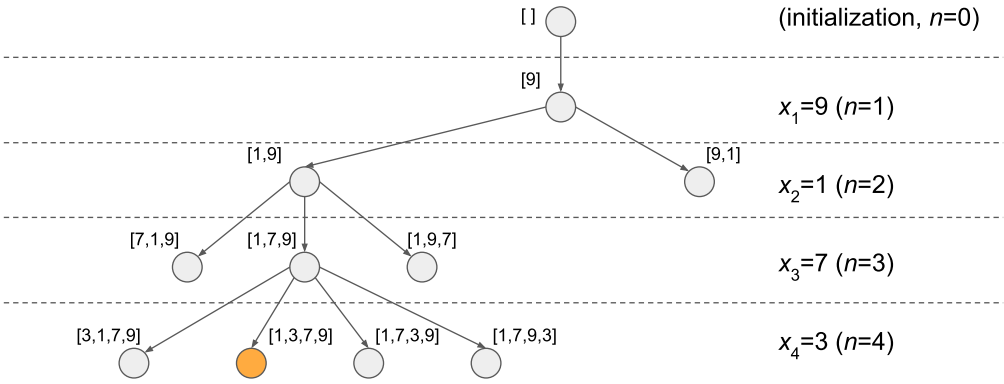

Figure 4.3.: Computational graph of greedy SDP insertion sort, solving the sorting problem for the input data set $\{9, 1, 7, 3\}$.

Section 5.

Dynamic programming

Relevant reference reading material for this section are **CLRS**, Chapter 14.

5.1. Efficient factorization and the principle of optimality

Dynamic programming (DP) is a sophisticated SDP algorithm strategy which is often able to efficiently solve difficult combinatorial AI problems in polynomial computational complexity or better. As with all SDP algorithms, the efficiency of DP depends heavily on the ability to **factorize** the generation of all possible configurations to reduce the number of computational steps required. Many combinatorial factorizations are known; for instance we have seen how the generation of permutations can be performed by successive insertion which means the permutation SDP takes just $O(N)$ steps.[1] DP combines this efficient SDP factorization with the **principle of optimality**, which exploits the algebraic principle of **distributivity**, to avoid the need to retain only the current optimal configuration. A couple of examples,[2]

$$\min(a+b, a+c) = a + \min(b, c) \tag{5.1}$$
$$\max(a \times b, a \times c) = a \times \max(b, c), \tag{5.2}$$

For objective functions which are **linear functions** of the components of the configuration, we can apply the above distributivity property to derive the corresponding **Bellman recursion** for the DP algorithm

$$F(X_n^\star) = \min_{X' \in S_{n-1}} F(X'), \tag{5.3}$$

where S_{n-1} contains extensions of the optimal configuration $X_{n-1}^\star$, accompanied by the recursion for the optimal configuration itself,

$$X_n^\star = \arg\min_{X' \in S_{n-1}} F(X'), \tag{5.4}$$

for all $n = 1, 2, \ldots, N$. To get the recursion started at $n = 0$, we need the root configuration $X_0^\star$ and $F(X_0^\star)$; their values depend upon the specific DP algorithm.

It helps to explain (5.3) in a bit more detail. At each stage n, the candidate configurations are obtained by extending the optimal configuration $X_{n-1}^\star$ in the previous stage, so the optimal value of the objective function $F(X_n^\star)$ and corresponding optimal configuration $X_n^\star$, can be obtained using only the optimal configuration from the previous stage.[3] If the factorized SDP takes $O(N)$ steps and if there are at most k ways to extend each configuration, the complete DP algorithm has time complexity

[1] Note that this does not mean that the corresponding space complexity of this factorization is linear in N. Indeed, the for permutations the space complexity grows with $N!$.

[2] This is exactly the same algebraic principle as $ab + ac = a \times (b + c)$ but in a different algebra!

[3] This may help to explain why (5.3) is a *recursive equation*.

$O(Nk)$ i.e. it is linear in the size of the input. This is usually remarkably efficient because k is often small, and DP is therefore used widely in AI and ML.

Of course, whether it is possible to find a factorization with small k and thus a fast Bellman recursion for a particular AI optimization problem is not an easy problem in itself to solve, and requires some ingenuity to devise.

5.2. Example: maximum sum tail subsequences

As an example of using DP in practice, let us look at an example of the **maximum sum tail subsequence** problem. The tail subsequences of a list with three items $[x_1, x_2, x_3]$ are $[x_3]$, $[x_2, x_3]$ and $[x_1, x_2, x_3]$. The problem is to (efficiently) find the tail subsequence with maximum sum, i.e. for three items, select between x_3, $x_2 + x_3$ and $x_1 + x_2 + x_3$. This problem arises in logistics: if we have a queue of containers at a dock to load onto a cargo ship which must be loaded in order, and shipping each container leads to either a profit for the the shipping company (positive value data item) or loss for the company (negative value item), then the shipping company will want to maximize the profit by emptying the optimal amount of the queues in the container, onto the ship.

To express the optimization problem mathematically, we need a representation of the load (a configuration), X. Here we use lists, so $X = [4, 5, -3]$ is a load selecting one of the tail subsequences of the input list $[2, 7, 4, 5, -3]$. Using this representation, the optimization problem is,

$$X^\star = \arg\max_{X' \in \mathcal{X}} \left(\sum_{x' \in X'} x' \right), \tag{5.5}$$

where here $\mathcal{X}$ represents the set of all possible tail subsequences of the list $[x_1, \ldots, x_N]$. So, the problem is expressed as computing the sum of all possible subsequence loads and selecting the one with the largest value.[4]

To solve this using DP, we need an efficient factorized SDP for generating items in the set $\mathcal{X}$. This can be done using the following simple procedure: starting with the empty list $[\,]$, at each subsequent stage, either add an empty list $[\,]$ to the set of candidate configurations, or append the next item in the input x_n onto the end of all current candidate lists (Figure 5.1). For DP to be applicable, we also require the objective function to be linear in the configuration update. In this case, appending an item to a configuration corresponds to adding the value of this item to the total objective function value for that configuration, which is a linear operation as required.

Finally, since an empty load will have zero profit/loss, this provides an enough information to write down the Bellman recursion for the optimal profit of the shipping company,

$$P_n^\star = \max\left(0, P_{n-1}^\star + x_n\right), \tag{5.6}$$

for $n = 1, 2, \ldots, N$ with $P_0^\star = 0$ to start the recursion at $n = 0$. The corresponding optimal choice of list of containers to load is given by,

$$X_n^\star = \begin{cases} [\,] & P_n^\star + x_n \leq 0 \\ \text{append } x_n \text{ to the end of } X_{n-1}^\star & \text{otherwise.} \end{cases} \tag{5.7}$$

See Figure 5.2 for a toy example illustrating the sequence of steps in the above Bellman recursion. In terms of computational complexity, this algorithm runs in $O(N)$ time, whereas, to solve the problem

[4] For consistency here we could also minimize the *negative sum* of values, but this is entirely equivalent to *maximizing the positive sum*.

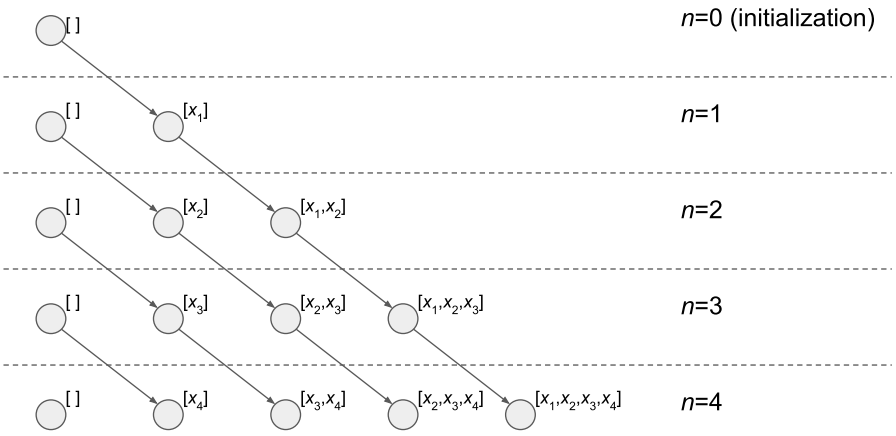

Figure 5.1.: Computational graph of a fully exhaustive sequential decision process (SDP) for generating all tail subsequences of a length four input sequence $[x_1, \ldots, x_4]$.

exhaustively would require $O\left(N^2\right)$ complexity. This shows how DP can lead to very efficient solutions to discrete optimization problems in AI.

Learning outcomes for section 5

- ▷ Explain the principles of dynamic programming (DP) for exactly solving combinatorial optimization problems in AI.
- ▷ Recognize, design and apply an appropriate Bellman recursion for specific DP algorithms.
- ▷ Determine the computational complexity of a DP algorithm.

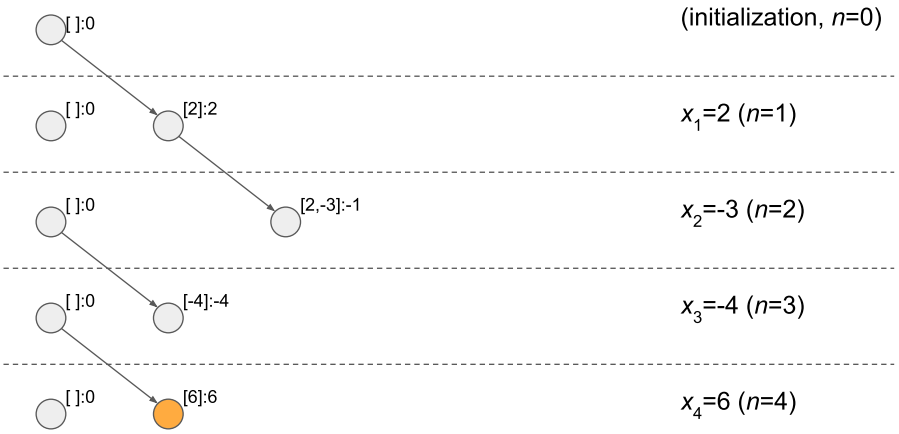

Figure 5.2.: Computational graph of DP for solving the maximum sum tail subsequence, for the input data list $[2, -3, -4, 6]$. The numbers after the subsequences refer to the sum of that subsequence, e.g. $[2, -3] : -1$ means that the subsequence $[2, -3]$ has total sum -1. The optimal solution is found at the final stage, $X^\star = [6]$ with $F\left(X_4^\star\right) = 6$.

Section 6.

Approximate combinatorial algorithms

Relevant reference reading material for this section: **R&N**, Section 4.1 and **MLSP**, Section 2.6.

6.1. Global and local optima

Exact optimization methods are completely reliable, but typically quite slow, whereas **approximate methods** can be fast but not necessarily reliable. Approximate methods cannot guarantee finding an optimal solution to (3.1), only a solution which is 'good enough' for a specific purpose. Various algorithm strategies for approximate optimization exist: **stochastic** (randomized), **greedy**[1], **tabu search**, **genetic optimization**, **particle swarm optimization** (PSO), **simulated annealing**, **local beam search** and many others. Approximate methods usually start with some guess configuration and aim to find a configuration which has an objective function value which is at least better than the starting guess.

Without additional information about a specific problem, a few things can be said in general about approximate optimization methods (see Figure 3.1):

1. In the search for an improvement over the starting configuration, they can become "trapped" in local optima which are indistinguishable from global optima,

2. The success of any approximate algorithm can be heavily dependent upon the starting guess: a good guess near to the global optima will be better than a bad guess close to a local optima and/or far from the global one, however, since there is no way in general of discriminating global from local optima we cannot know whether the starting guess is good or bad (without comparing different starting guesses),

3. It is usually not possible to know how good a local optima is, by comparison to the (unknown) global optima.

4. If the algorithm stops at an optima, this may actually be the global optima, but there is no way to know this in general.

Some sophisticated stochastic algorithms have **convergence guarantees**, for example, **Markov chain Monte Carlo** (**MCMC**) methods such as **simulated annealing** (which we will investigate below) but only in the limit of an infinite number of iterations. Even in this situation, the number of iterations required to find a configuration whose objective is within some required tolerance of the optimal solution, is typically unknown.

[1] Here greedy refers to *approximate greedy*, not *exact greedy*.

Algorithm 6.1 Approximate greedy search for combinatorial optimization of AI problems.

▷ **Step 1**. *Initialization*: Start with iteration $n = 0$, select an initial candidate configuration X_0, choose a maximum number of iterations R.

▷ **Step 2**. *Neighbourhood search*: Find the configuration X' within the configuration neighbourhood $\mathcal{N}(X_n)$ with smallest objective function value $F(X')$, and set $X_{n+1} = X'$. If $F(X') \geq F(X_n)$, then terminate with $X^\star = X_n$.

▷ **Step 3**. *Iteration*: Set $n = n + 1$, and while $n \leq R$, go back to step 2, otherwise exit with solution $X^\star = X_n$.

6.2. Approximate greedy search

In this section we will explore a simple but often surprisingly effective, approximate method known as **greedy approximate search**, **hill-climbing** or **iterative improvement** (Algorithm 6.1). It is based on the idea of performing many searches which are optimal within a certain neighbourhood of the current configuration, and using the best known local configuration to centre the next local search. When this algorithm becomes trapped at some optima in the objective function (as it will eventually), it terminates with the last best known configuration.

To use the algorithm we need a definition of **configuration neighbourhood**, $\mathcal{N}(X)$ of some configuration X. There are many ways to measure distance (the mathematical name for which is **metric**) between configurations, one widely used metric is the **Hamming distance** $d(X, X')$ which counts the number of elements of the two configurations X and X' which differ. For instance, if the configurations are permutations, then $d(X, X')$ would measure the number of places in which the permutations disagree; when they are the same permutation then $d(X, X) = 0$. Thus a combinatorial neighbourhood based on this metric[2] would be defined mathematically as

$$\mathcal{N}(X) = \{X' \in \mathcal{X} : d(X, X') \leq r\}, \tag{6.1}$$

for some given maximum Hamming distance $r > 0$.

Some facts about the behaviour of this algorithm can be inferred directly. Firstly, by design, the sequence of candidate configurations at each iteration $X_0, X_1, X_2, \ldots$ will have decreasing objective function value, i.e. $F(X_{n+1}) < F(X_n)$ for all $n = 0, 1, 2, \ldots$. Because the algorithm is approximate, there is no guarantee that the final value $X^\star$ coincides with the global minima. The algorithm could finish before $n = R$, but we cannot predict in advance when it will terminate. There is trade off between the neighbourhood size r, and the efficiency of the algorithm. If r is large, the search will be slow, but since each iteration covers more of the space $\mathcal{X}$, it is more likely to find the global optima. If r is small, each iteration will be fast, but the more likely the algorithm will be to get trapped in local optima and never find the global optima.

6.3. Example: maximum sum combinations

To illustrate greedy search, we will investigate the **maximum sum combination problem**, that is, finding a size-M combination of the N input data items, such that the sum of this combination is maximized. For instance, with input data $x = [0.5, -0.1, 0.3, -0.2]$, the optimal size 2 combination is

[2] Or any other combinatorial metric, in fact.

Iteration	**Configuration** X_n	**Objective value** $F(X_n)$
$n = 0$ (initialize)	$\{0.33, -0.19, -0.59, -0.14\}$	-0.59
$n = 1$	$\{0.33, -0.19, \mathbf{2.18}, -0.14\}$	2.18
$n = 2$	$\{0.33, \mathbf{1.19}, 2.18, -0.14\}$	3.56
$n = 3$	$\{\mathbf{1.19}, 1.19, 0.33, 2.18\}$	4.89
$n = 4$ (terminate)	$\{1.19, 1.19, \mathbf{1.07}, 2.18\}$	5.63

Table 6.1.: Example output from approximate greedy search, Algorithm 6.1, applied to the maximum sum combination problem, for input data size $N = 40$ and combination size $M = 4$. The neighbourhood search uses Hamming distance with radius $r = 1$. In this run, the globally optimal solution is found by the algorithm after 4 iterations. The **bold** numbers at each stage highlight values from the input which are replaced during each stage. Input data values are x = [-0.43, -1.67, 0.13, 0.29, -1.15, 1.19, 1.19, -0.04, 0.33, 0.17, -0.19, 0.73, -0.59, 2.18, -0.14, 0.11, 1.07, 0.06, -0.10, -0.83].

$\{0.5, 0.3\}$ with sum 0.8.[3] This problem comes up in many real-world situations. As an example, in graphic design, you have to fill all of the advertising slots of a magazine with advertisements. There are N advertisers and the profit the publisher makes from the n-th sponsor is x_n (some advertisements are *loss leaders* – a small loss is made on these – but the publisher is forced to fill out every advertising page in the magazine). Maximizing profit from the magazine is a matter of solving an instance of the maximum sum combination problem. The problem is specified mathematically as,

$$X^\star = \arg\max_{X' \in \mathcal{X}} \sum_{x' \in X'} x', \tag{6.2}$$

where $\mathcal{X}$ is the set of all size M combinations of the data items $x_1, \ldots, x_N$.

For the search neighbourhood, we will use the Hamming distance which in this case is the number of items in two combinations X and X' which differ. To see how good the result of this approximate search is, for small N and M at least, we can find the global optima using exhaustive search. Computational experiments confirm that for small N and M, for Hamming distance $r = 1$, approximate greedy search strategy apparently *almost always* finds the global maxima in a handful of iterations (see Table 6.1).

From these experiments it seems that hill-climbing is a good solution to the problem. However, in fact this problem can be solved exactly by selecting the top M items in the input data, which requires sorting the data with worst case computational complexity of $O(N \log N)$. So, the *computational efficiency* of Hamming-distance based greedy approximation for this particular problem is doubtful: its success comes at the expense of very slow neighbourhood search. We can speed up the neighbourhood search by selecting a subset of the neighbourhood $\mathcal{N}(X_n)$ at each iteration; when we do this with a random selection of k candidate configurations for instance, the algorithm *almost always fails* to find the globally optimal solution, although the solution is often quite close to optimal.

[3] This is a simplified instance of the more complex **knapsack** or **bin-packing** problem.

Learning outcomes for section 6

- ▷ Identify and explain the implications of local and global optima in combinatorial optimization for problems in AI.
- ▷ Provide the steps in the approximate greedy search algorithm and apply the algorithm to combinatorial optimization problems.
- ▷ Determine the computational complexity of a specific application of approximate greedy search.

Section 7.

Simulated annealing

The recommended reference reading material for this section is **R&N**, Section 4.1 and **MLSP**, Section 2.6.

7.1. Escaping local optima

Approximate greedy search, Algorithm 6.1, is not guaranteed to find the global optima, because it can become trapped in local optima. This happens because it only improves the value of the objective at each iteration. If we could escape these local optima, the algorithm could (in principle) find the global optima, at least eventually. **Simulated annealing** (Algorithm 7.1) is an approximate search method which can accept an updated configuration with *worse* objective function value.[1]

The probability of accepting a 'bad' step is inspired by the **Boltzmann distribution** from the discipline of thermodynamics:

$$\Pr\left(\text{accept}\right) = \exp\left(-\left[F\left(X'\right) - F\left(X_n\right)\right]\frac{n}{kR}\right). \tag{7.1}$$

The worse $F\left(X'\right)$ is in the search neighbourhood $\mathcal{N}\left(X_n\right)$, the smaller the probability of accepting this bad step (the algorithm tries to take the *least worst* bad steps). Furthermore, as n gets closer to the maximum number of iterations, R, the probability of accepting a bad step decreases. This means that simulated annealing eventually behaves like simple approximate greedy search, only improving the objective, and finally converging on a value $X^\star$ (which is not guaranteed to be optimal). The

[1] The name of this method comes from the physical manufacturing process of slowly cooling an amorphous crystalline solid (such as glass), to strengthen it by removing structural defects.

Algorithm 7.1 Simulated annealing for combinatorial optimization of AI problems (here stated for the case of objective function minimization rather than maximization).

- ▷ **Step 1**. *Initialization*: Select a starting candidate configuration X_0 and set $X^\star = X_0$, set iteration number $n = 0$, set $F^\star = \infty$, choose annealing schedule $k > 0$, and maximum number of iterations R
- ▷ **Step 2**. *Neighbourhood search*: Choose any configuration X' from $\mathcal{N}\left(X_n\right)$. If $F\left(X'\right) < F\left(X_n\right)$, set $X_{n+1} = X'$, otherwise choose a random number q from $[0, 1]$, and if $q > \exp\left(-\left[F\left(X'\right) - F\left(X_n\right)\right]\frac{n}{kR}\right)$ then set $X_{n+1} = X'$
- ▷ **Step 3.** *Maintain best configuration*: If $F\left(X_{n+1}\right) < F^\star$, set $F^\star = F\left(X_{n+1}\right)$ and $X^\star = X_{n+1}$
- ▷ **Step 4**. *Iteration*: Set $n = n + 1$, and while $n < R$, go back to step 2, otherwise exit with solution $X^\star$.

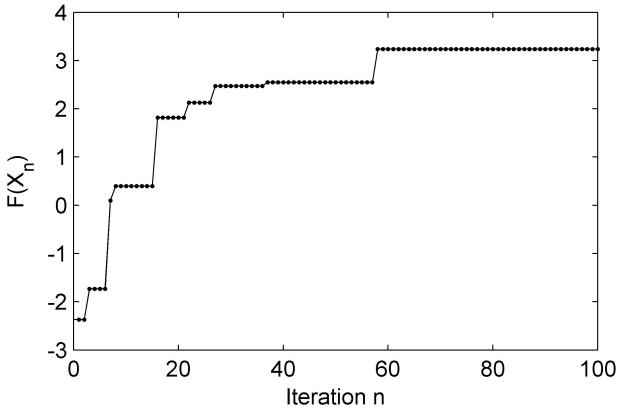

Figure 7.1.: Simulated annealing applied to the maximum sum combination problem. Here, the data has $N = 20$ items and the configurations are combinations of size $M = 4$. The annealing schedule was $k = 0.1$ and the maximum number of iterations $R = 100$. On this run, the iteration converges on the exact global configuration.

parameter $k > 0$ is known as the **annealing schedule**. If k is large, the probability of taking a bad step decreases slowly and convergence will take a large number of iterations. Conversely, if k is small, convergence is rapid but the algorithm is more likely to get trapped in a local optima, behaving essentially as approximate greedy search.

7.2. Example: annealing better maximum sum combinations

In the previous section, we observed that the Hamming distance neighbourhood search for the approximate greedy algorithm is accurate (i.e. mostly able to find the global optima) but intractable, but that when the neighbourhood size was reduced to make the algorithm more tractable, the algorithm was no longer accurate. The reason for this loss of accuracy is that the reduced neighbourhood search causes the greedy algorithm to get stuck. Simulated annealing has much simpler neighbourhood search which only requires selecting one configuration, and can avoid getting stuck through backwards steps. Although taking very many more iterations than approximate greedy search, computational experiments on the same problem and same data as approximate greedy search, is quite effective and often converges on the exact global solution (Figure 7.1).

We can see from the run shown in Figure 7.1 that although simulated annealing tends to get trapped in local maxima however, it eventually escapes these local maxima to quickly find a new, better configuration, eventually reaching the global optima.

Learning outcomes for section 7

- ▷ Explain the steps in the simulated annealing algorithm.
- ▷ Compare the advantages and disadvantages of simulated annealing algorithm for approximate global optimization of AI problems.

Section 8.

Logic

Relevant reference reading material for this section is **R&N**, Section 7.3 and Section 7.4.

8.1. Propositional calculus

Any computational system is, at root, an application of **logic** to data about problems in the real world. This application of logic is essential to any form of **rational decision-making** and **reasoning**, and AI systems must be rational if they are to be reliable. **Logical inference** involves combining facts about the world using the rules of logic, to draw correct conclusions on other facts about the world. For instance, the chain of common-sense reasoning which allows us to combine the rule 'if it is sunny then it is hot' with the fact that today is sunny, to conclude that today is hot, is an example of rational logical inference.

In order to automate logical reasoning like this computationally, we use **propositional calculus** to formalize the process mathematically. This is a type of **algebra** with extremely similar algebraic rules to the rules of algebra in ordinary arithmetic computations. The **syntax** of the algebra, that is, the rules by which the symbols in the algebra are combined, is identical to that in ordinary algebra, but instead of using plus, times and ordinary numbers, makes use of special symbols for **logical operations** or logical **connectives**, see Table 8.1. For example, the **logical expression** (**sentence**) '$True \vee (x \wedge y)$' is syntactically correct (well-formed), but '$\vee \wedge True)(xy$' is not, in exactly the same way that '$1 \times (x + y)$' is a valid algebraic expression 'but $+ \times 1)(xy$' is not.

Unlike arithmetic which uses ordinary numbers, the **semantics** of variables and their values in propositional calculus are **Boolean**, that is, they can only take on one of the **truth values** $True$ or $False$. This means that all the functions which operate on those variables are Boolean functions, which can only take and output Boolean values. The semantics of Boolean functions can therefore be explicitly listed using **truth tables**, see Table 8.1 for the truth tables of the logical connectives. The possible truth values of an expression in propositional calculus can thus always be given using truth tables.

Propositional variables, which represent some state of the world, are called **atomic (logical) statements** or **simple** and are denoted in expressions by the upper case italic variable names P, Q, R etc. Examples: $R =$ 'raining today' or $D =$ 'the car is in the driveway'. The Boolean **(logical) constants** $True$ and $False$ are also atomic statements. Expressions which are made by combining two or more atomic expressions using connectives are called **complex statements** or **compound statements**. Examples of compounds statements are $R \wedge \neg D$ which is interpreted as 'it is raining today *and* the car is *not* in the driveway'.

If we have the truth values of all the variables in a propositional expression, then we can **evaluate** the expression to find its truth value. We do this by replacing the syntactic elements by their semantic counterparts, then using truth tables recursively until we have a single truth value. This is exactly the same process as performing arithmetic in order to evaluate an algebraic expression, but using logical

Symbol	**Meaning**	**Truth table**
$\neg$	Not (negation)	x | $\neg x$ $False$ | $True$ $True$ | $False$
$\wedge$	And (conjunction)	x | y | $x \wedge y$ $False$ | $False$ | $False$ $False$ | $True$ | $False$ $True$ | $False$ | $False$ $True$ | $True$ | $True$
$\vee$	Or (disjunction)	x | y | $x \vee y$ $False$ | $False$ | $False$ $False$ | $True$ | $True$ $True$ | $False$ | $True$ $True$ | $True$ | $True$
$\Longrightarrow$	If ... then (implication)	x | y | $x \Longrightarrow y$ $False$ | $False$ | $True$ $False$ | $True$ | $True$ $True$ | $False$ | $False$ $True$ | $True$ | $True$
$\Longleftrightarrow$	If and only if (equality)	x | y | $x \Longleftrightarrow y$ $False$ | $False$ | $True$ $False$ | $True$ | $False$ $True$ | $False$ | $False$ $True$ | $True$ | $True$

Table 8.1.: The logical connectives of propositional calculus.

connectives rather than the usual arithmetic operations. For example, if $H =$ 'the house is occupied' and we have the proposition $\neg R \wedge \neg D \implies \neg H$, and if in the real world, $R = True$, $D = True$ and $H = True$, we can calculate,[1]

$$\begin{aligned} \neg R \wedge \neg D \implies \neg H =& \neg True \wedge \neg True \implies \neg True \\ =& False \wedge False \implies False \\ =& False \implies False \\ =& True, \end{aligned} \tag{8.1}$$

so we can conclude that under these specific conditions, the proposition holds (i.e. it is true). We can interpret this as the claim that 'if it is not raining today and the car is not in the driveway then the house is empty', is true in this case.

A set of conditions like this in the real world which give assignments of actual truth values to variables, is called a **(logical) model**. If some proposition A is true when the variables are set to model M, then we say that M **satisfies** A. The set of all models under which A is true is written $M(A)$. A very simple example: for the proposition $P = A \vee True$, the set of models $M(P) = \{A = False, A = True\}$.

8.2. Logical inference

The main purpose of propositional calculus is to use it to automate the process of logical reasoning, i.e. making correct logical inferences. We want to show that a proposition Q is a *necessary consequence* of P, or equivalently P **entails** Q,

$$P \vDash Q \text{ if and only if } M(P) \subseteq M(Q). \tag{8.2}$$

In words (8.2) means that every model in which P is true, is also true for Q.

Assume we have some set of facts and logical rules, which we will represent as propositions, which we will call a **knowledge base**, which we will give the label KB, the truth value of which is just the conjunction of all the propositions it contains. We want to be able to automate the process of deciding whether a proposition Q is a necessary consequence of KB, i.e. to test whether $KB \vDash Q$. The simplest way to do this is just to calculate (8.2) directly: compute $M(KB)$ and $K(Q)$ and if $M(KB) \subseteq M(Q)$ then our proposition is a necessary consequence of the knowledge base. This procedure, called **model checking** (Algorithm 8.1), is obviously perfectly reliable (we say the procedure is **sound and complete**), because it cannot fail to correctly identify entailment where it holds. We can see this as being a case of *exhaustive search* because it solves the problem by testing all possible configurations of the variables. Because of this, the complexity of model checking is exponential in N, the number of variables, in fact one has to check exactly 2^N possible combinations.

8.3. Example: automated medical decision-making

As an example of automated propositional reasoning, consider the (realistic but heavily simplified) Parkinson's medical knowledge base. It contains the following facts, $C =$'Satisfying diagnostic criteria', $O =$ 'Over age 60' and $P =$ 'Suspect Parkinson's disease', along with the *differential diagnostic rule* that satisfying the diagnostic criteria and being over 60, implies a suspicion of Parkinson's, $D =$

[1] Like algebra, there are *precedence* rules to remove ambiguity: first evaluate *not*, then *and*, then *or*, then *implication*, then *equality*. Brackets override the ordering.

Algorithm 8.1 Logical model checking for testing entailment $P \vDash Q$.

▷ **Step 1**. *Exhaustive search*: For all propositional variables A, B, C etc. upon which propositions P and Q depend, evaluate P and Q under all possible combinations of truth values of those variables.

▷ **Step 2**. *Compute models*. Using this find the set of models $M(P)$ where P holds and $M(Q)$ where Q holds.

▷ **Step 3**. *Test subset*: If $M(P) \subseteq M(Q)$ then return $P \vDash Q$ is true, false otherwise.

C	O	P	$(C \wedge O) \implies P$	$\neg O \implies \neg P$
$False$	$False$	$False$	$True$	$True$
$False$	$False$	$True$	$True$	$False$
$False$	$True$	$False$	$True$	$True$
$False$	$True$	$True$	$True$	$True$
$True$	$False$	$False$	$True$	$True$
$True$	$False$	$True$	$True$	$False$
$True$	$True$	$False$	$False$	$True$
$True$	$True$	$True$	$True$	$True$

Table 8.2.: Model checking applied to a simplified Parkinson's disease diagnostic knowledge base KB with the facts C ='Satisfying diagnostic criteria', O = 'Over age 60' and P = 'Suspect Parkinson's disease', along with the *differential diagnostic rule* that satisfying the diagnostic criteria and being over 60, implies a suspicion of Parkinson's, $KB = D = (C \wedge O) \implies P$. The query $Y = \neg O \implies \neg P$ asks whether being under 60 implies no suspicion of Parkinson's. The knowledge base cannot tell us this because $KB \vDash Y$ does not hold.

$(C \wedge O) \implies P$. There are $2^3 = 8$ possible models, so it is possible to apply Algorithm 8.1 by hand. We would like to know if the knowledge base $KB = D$ entails whether being under age 60 removes the suspicion of Parkinson's, which expressed in terms of the knowledge base is the proposition $Y = \neg O \implies \neg P$. Table 8.2 checks $KB \vDash Y$, and we can see from this that $M(KB)$ is true in cases where $M(Y)$ is not, thus it is not true that $M(KB)$ is contained in $M(Y)$ so the inference that being under 60 implies no suspicion of Parkinson's, does not follow from the knowledge base. Indeed, this mirrors what is known in the real world; although rare, there exist so-called *young onset* Parkinson's cases, a famous example being the actor Michael J. Fox.

Learning outcomes for section 8

- ▷ Explain the syntax and semantics of propositional calculus.
- ▷ Use the rules of propositional calculus to calculate the truth value of simple propositional expressions.
- ▷ Explain the model checking algorithm, and apply it to test entailment in simple logical knowledge bases.

Part III.

Machine learning

Section 9.

Machine learning and gradient descent

Recommended reference reading material for this section is **MLSP**, Section 2.3, Section 6.4, **R&N**, Section 18.6, **PRML**, Section 3.1 and **H&T**, Section 3.2. You should also read **Gill**, Sections 5.1-5.4 for the essential mathematics of the derivative and gradient of a function.

9.1. Overview of machine learning

Whereas symbolic AI deals with certainties and exact computational rules, **machine learning** (**ML**) extracts "*learns*")[1] patterns from data which may be uncertain and uses these to make **predictions**. It is therefore a form of **inductive reasoning**. Given some **training data**, ML algorithms are **trained** by minimizing an **error (objective, loss) function** $F(w)$ which depends upon the continuous w parameters of the ML model (along with some possibly discrete-valued parameters); the smaller the error function on the training data, the better. It then uses the trained parameters $w^\star$ in the model, to make predictions about new unseen data, the **test data**. ML algorithms are usually categorized according to the availability of labeled data: **supervised**, **unsupervised**, **self-supervised**, **transfer learning** and many others.

9.2. Training ML algorithms using sequential gradient descent (SGD)

Estimating the best model parameters in a ML problem, involves optimizing the error function $F(w)$ with respect to the continuous parameters w of the algorithm over the space of all possible algorithm values, $\mathcal{W}$:

$$w^\star = \arg\min_{w' \in \mathcal{W}} F\left(w'\right). \tag{9.1}$$

Typically, the space $\mathcal{W}$ is just the **Euclidean space** $\mathbb{R}^D$, where D is the number of parameter dimensions. Typically, the objective function (9.1) is not convex and has a great many local optima, so the training method does not necessarily seek to find the global optima for (9.1).[2]

One of the most widely used methods for attempting to solve (9.1) is **sequential gradient descent (SGD)** (Algorithm 9.1). This is a general algorithm for finding a value of the ML model parameters w such that error function $F(w)$ is as small as possible, expressed as a condition on the **multivariable gradient**, $F_w(w) = 0$[3], where F_w is the **partial derivative** of F with respect to the parameter vector w,

[1] It is important to note that this bears very little resemblance to the way humans and animals actually learn.
[2] Although modern ML algorithms are capable of finding parameters which find the value of w such that $F(w) = 0$.
[3] Note that this is actually a *vector* of D zeros, but we use the usual zero as a shorthand.

Algorithm 9.1 Sequential gradient descent (SGD) for optimization of model parameters in machine learning.

- ▷ **Step 1**. *Initialization*: Select a starting candidate w_0, set iteration number $n = 0$, choose convergence tolerance $\epsilon > 0$ and learning rate $\alpha > 0$
- ▷ **Step 2**. *Gradient descent step*: Compute new model parameters, $w_{n+1} = w_n - \alpha F_w(w_n)$
- ▷ **Step 3.** *Convergence test*: Compute new error function $F(w_{n+1})$ and loss function improvement $\Delta F = |F(w_{n+1}) - F(w_n)|$, and if $\Delta F < \epsilon$, exit with optimal solution $w^\star = w_{n+1}$
- ▷ **Step 4**. *Iteration*: Set $n = n + 1$, go back to step 2.

$$F_w(w) = \begin{bmatrix} \frac{\partial F}{\partial w_1}(w) \\ \frac{\partial F}{\partial w_2}(w) \\ \vdots \\ \frac{\partial F}{\partial w_D}(w) \end{bmatrix}, \tag{9.2}$$

which is just the vector of gradients in each dimension separately.

The basic idea is this: starting with a guess for w_n, take a "step" in the direction of **steepest descent** of the loss function, $-F_w(w)$, and take this as a better guess w_{n+1}. The size of the step, $\alpha > 0$, is called the **learning rate** which determines how quickly the minimum is reached. This simple procedure is, of course, not guaranteed to find the global minimum, unless the error function is convex with respect to w which with most ML algorithms is generally not the case. More advanced versions of this algorithm are very widely used in modern ML.

One of the difficulties with SGD is that it is possible to *overshoot* a minimum if the learning rate is too large. As a result the error function on the next iteration can be larger than on the previous one. The next step occurs may be in nearly the opposite direction, potentially overshooting again, and so on. This is the reason for computing the **absolute value** $||$ of the improvement, ΔF: if we only computed the difference in loss function values at each step, we could end up with negative improvement and never terminate. Furthermore if α is above a critical threshold the algorithm might **diverge**, that is the objective function increases without bound. Choosing the learning rate parameter is, therefore critical and often more of an art than a science with complex modern ML algorithms.

9.3. Example: linear regression

A **regression model** learns to fit a **curve** $f(x, w) = y$ through pairs of training data (x_n, y_n) for $n = 1, 2, \ldots, N$ where w are the model parameters. Once fitted to the training data, we can use the best parameters $w^\star$ and some input data x, to make predictions y. The tacit hypothesis is that there is some (*continuous*) *relationship* between input and output variables x and y. This is a supervised technique: given input data and corresponding labeled output data, regression attempts to find a curve which fits the training data best.

The specific hypothesis depends upon the chosen form of regression function. For instance, a linear model for $D = 1$ is a *straight line*, and for $D > 1$ a **hyperplane**. This is called a **linear regression** model,

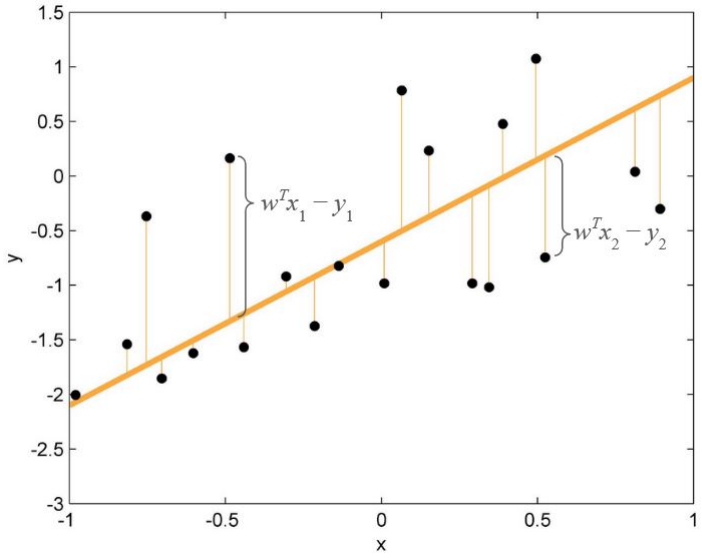

Figure 9.1.: Linear regression in $D = 2$ (with first coordinate values set to 1) as single dimension curve fitting, showing the interpretation of the sum-of-squares error $F(w) = \sum_{i=1}^{N} \left(w^T x_i - y_i\right)^2$, where the terms $w^T x_i - y_i$ are the *perpendicular distances* between the best fit line (orange) and the coordinates of the data points (black dots).

$$\begin{aligned} f(x, w) &= w_1 x^1 + w_2 x^2 + \cdots + w_D x^D \\ &= w^T x, \end{aligned} \tag{9.3}$$

where we use superscript to index each dimension (row) of the vector x.[4] Using this, we can write down the **square loss function**, which is the square of the difference between the model prediction $w^T x_n$ on each data item x_n and the corresponding labeled item in the data, y_n,

$$F(w) = \sum_{i=1}^{N} \left(w^T x_i - y_i\right)^2, \tag{9.4}$$

which quantifies the quality of the fit of the linear regression model to the data. In $D = 2$ dimensions, if we fix the first data coordinate to 1 (which becomes the **intercept** of the line on the plot), we can visualize the model fit on an x-y plot which shows that the error is the sum of the squares of the perpendicular distances to the best fit line (Figure 9.1).

As an example, let us apply SGD to this loss function. To do this, we need the gradient of (9.4), which is,[5]

$$F_w(w) = 2 \sum_{i=1}^{N} \left(w^T x_i - y_i\right) x_i. \tag{9.5}$$

[4] The *vector transpose* notation x^T turns the row vector x into a column vector, so we can use vector-vector multiplication leading a more compact form of the linear equation.

[5] The proof of this is straightforward but takes quite a lot of tedious algebra. Alternatively, we can use standard results from matrix-vector calculus.

Iteration	**Parameters** w_n	**Error function value** $F(w_n)$	**Improvement** ΔF
$n = 0$ (initialize)	$[-0.09, -1.60]$	69.24	N/A
$n = 1$	$[0.10, -0.07]$	28.20	41.04
$n = 2$	$[-0.41, 0.60]$	16.46	11.73
$n = 3$	$[-0.28, 1.04]$	13.08	3.38
$n = 4$	$[-0.47, 1.22]$	12.10	0.99
$n = 5$	$[-0.40, 1.35]$	11.80	0.29
$n = 6$	$[-0.48, 1.39]$	11.71	0.09
$n = 7$ (exit)	$[-0.44, 1.43]$	11.68	0.03

Table 9.1.: Example output from sequential gradient descent (SGD), Algorithm 9.1, applied to the linear regression problem for data of dimension $D = 2$, first coordinate fixed to 1, with input data size $N = 20$. Algorithm parameters were learning rate $\alpha = 0.08$ and convergence tolerance $\epsilon = 0.05$. In this run, the algorithm terminates at stage $n = 7$ with result $w^\star = [-0.44, 1.43]$ and $F(w^\star) = 11.68$, which compares favourably to the known optimal solution, $w = [-0.6, 1.5]$.

so that the parameter update in the gradient descent step in SGD is,[6]

$$w_{n+1} = w_n - \alpha \sum_{i=1}^{N} \left(w_n^T x_i - y_i \right) x_i. \tag{9.6}$$

Results of a typical run of SGD using this gradient update step is shown in Table 9.1, which terminates after eight iterations.

For linear regression, the error function is actually convex with respect to the parameter vector w, so SGD with the correct choice of learning rate parameter, can converge on the globally optimal solution eventually.[7]

9.4. Model complexity and generalization

The linear regression model investigated above is particularly simple, but for most practical ML problems linear modeling is too simple. More complex and sophisticated regression models include **polynomials** in the form of sums of powers of the input variables, **kernel methods**, or **neural networks** (which we will study in later sections). It is usually the case that a sufficiently complex model, can actually achieve *zero training data error*, that is, it can obtain $F(w^\star) = 0$ at the optimal $w^\star$ on the training data. While this may seem like a desirable property for a trained ML model, it is often the case that an algorithm which has very low training set error, will, paradoxically, have *high test data error*. The reason for this is simple: the measured data often has two parts to it, a **systematic component** (we choose the mathematical form of the regression model to match this) and a **random component** to it (which influences the way we quantify prediction error).[8] Increasing the model complexity means that the model mostly fits the random component, not the underlying,

[6] Because we have freedom to choose any learning rate parameter α', then we can set $\alpha = 2\alpha'$ and eliminate the factor 2 in the gradient.

[7] In fact, in the case of linear regression and sum-of-square error, we do not actually need to use SGD, there is a straightforward analytical solution obtained using matrix-vector calculus. Such analytical solutions do not exist for most ML models, however.

[8] This is not always true in practice, some modern ML problems and corresponding algorithms are better described as *interpolation* rather than *least-error curve-fitting*, in these circumstance *smoothness* of the model in between data points is a better way to measure model complexity.

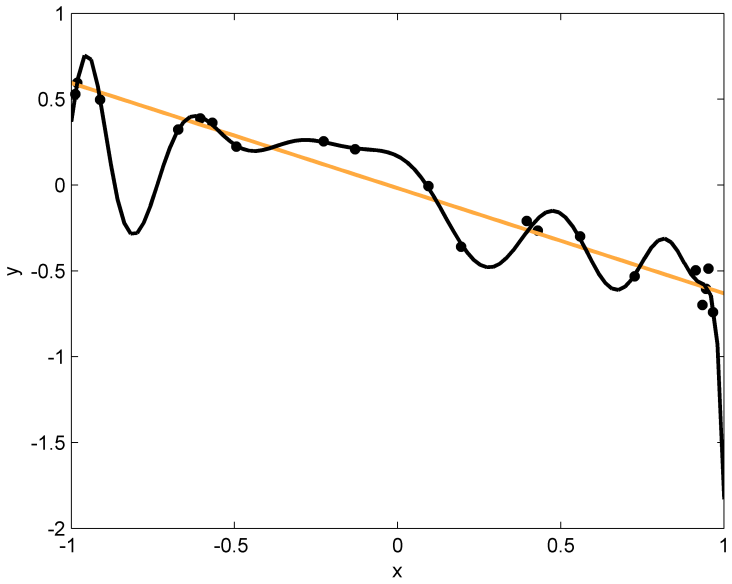

Figure 9.2.: Illustrating ML model complexity in regression. A complex polynomial regression model (black line) goes almost exactly through the training data (black dots), and thus has small training error, $F(w^\star) = 0.04$. By contrast, a simple linear model (orange) misses most of the black dots and so has much higher training error, $F(w^\star) = 0.17$. However, the actual model which generated this data is linear, the non-linear arrangement of the data is just pure randomness. Therefore, the complex polynomial model is *too complex* for this problem, even though the model training error is small.

systematic component (Figure 9.2). So, there is typically a trade off between model complexity and test error, and this is a common feature of most machine learning models. Ideally, we want a model which is as simple as possible, but no simpler.[9]

Learning outcomes for section 9

- ▷ Demonstrate understanding of the principles of machine learning: training versus test data, fitting models, making predictions, error functions, supervised versus unsupervised learning.
- ▷ Be able to apply sequential gradient descent to simple linear regression problems.

[9] This principle is often called *Occam's razor*.

Section 10.

Clustering

Relevant reference reading material for this section is **PRML**, Section 9.1, **MLSP**, Section 6.5 and **H&T**, Section 13.2.1 and Section 14.3.6.

10.1. Formulating optimal K-clustering

A common data analysis problem which arises in practice is clustering. Given N data points in D-dimensional data space, e.g. $\mathbb{R}^D$, what is the optimal way to **partition** the set of data into K groups? Clustering is an example of an **unsupervised** ML problem: we do not have any associated labels. As discussed in Section 3, we can always use an exhaustive combinatorial algorithm, i.e. find all possible ways of partitioning the set of indices $\{1, 2, \ldots, N\}$ into K blocks, then find the optimal partition. However, the number of such set partitions is the *Stirling numbers of the 2nd kind*, for K fixed this number increases like $O\left(K^N\right)$ (exponential) so exact combinatorial optimization algorithms are intractable. Therefore, in ML, approximate clustering methods are nearly always used in practice.

To solve the problem computationally, we need to formalize it mathematically. We will treat it as a **K-clustering** problem: partitioning the data into K non-overlapping clusters such that some objective function of this clustering configuration is minimized. We will use **indicator function** notation X_{ik}, to represent a clustering partition as a combinatorial configuration,

$$X_{ik} = \begin{cases} 1 & \text{data item } x_i \text{ is assigned to cluster } k \\ 0 & \text{otherwise} \end{cases}. \tag{10.1}$$

As an example, if we have data $x_1, \ldots, x_5$ and $K = 3$ classes, then the configuration,

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \tag{10.2}$$

implies x_1, x_2 are assigned to cluster 3, x_3 is assigned to cluster 1 and x_4, x_5 are assigned to cluster 2. This form is very convenient mathematically because we can then express the so-called **K-means objective**, a widely-used measure of clustering quality, directly in terms of this indicator function,

$$F(X, \mu) = \sum_{i=1}^{N} \sum_{k=1}^{K} X_{ik} \|x_i - \mu_k\|^2 . \tag{10.3}$$

where in this equation, $\| \|^2$ is the **square Euclidean norm** of a vector, which is just the sum of squares of each dimension,

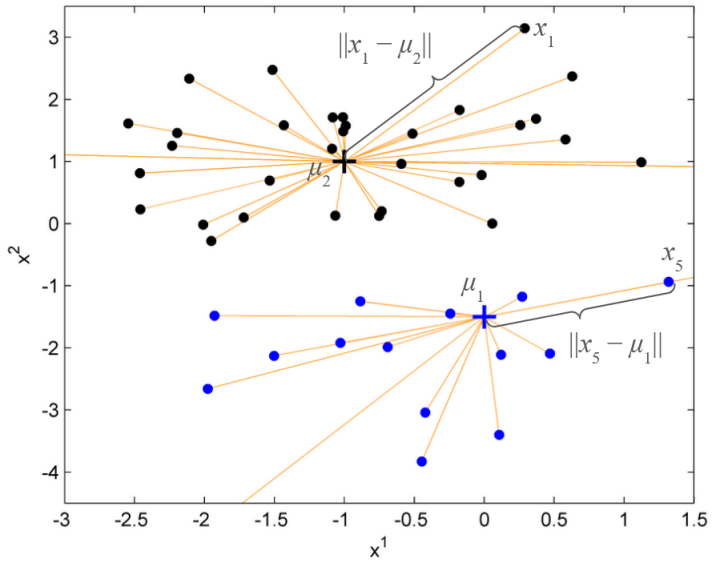

Figure 10.1.: Illustrating the objective function of K-means clustering in $D = 2$ and a partition of $K = 2$ clusters. This shows the measure of clustering quality $F(X, \mu) = \sum_{i=1}^{N} \sum_{k=1}^{K} X_{ik} \|x_i - \mu_k\|^2$ where the terms $\|x_i - \mu_k\|$ are the *Euclidean distances* between the coordinates of the cluster centroid μ_k (cross) and the coordinates of the data point x_i (dot). The colour (black, blue) indicates the assignment of the data point to clusters, given by the partition indicator (configuration) X_{ik}.

$$\|v\|^2 = \sum_{d=1}^{D} \left(v^d\right)^2. \tag{10.4}$$

The quantity μ_k is the *average* of the data in each coordinate of each cluster, which is called the **centroid** for that cluster, which we can also write as,

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N} X_{ik} x_i, \quad N_k = \sum_{i=1}^{N} X_{ik}.$$

It is helpful to visualize an example K-clustering in $D = 2$, showing how the objective is sensitive to the spread of the data around the centroids (Figure 10.1).

The K-means clustering problem thus attempts to solve the following optimization problem,

$$(X^\star, \mu^\star) = \underset{(X', \mu') \in \mathcal{W}}{\arg\min}\ F(X', \mu'). \tag{10.5}$$

where $\mathcal{W}$ is the set of all possible partitions (i.e. just those indicators where each data item is assigned to a unique class), along with their corresponding centroid averages.

10.2. The K-means algorithm

Note that, given fixed clustering configuration X, we can always find the globally optimal corresponding centroids μ, and given fixed μ, we can determine the globally optimal corresponding configuration

Algorithm 10.1 The K-means algorithm for approximate clustering.

- ▷ **Step 1**. *Initialization*: Start with $n = 0$ and an initial guess for all centroids μ_k^0 for $k = 1, 2, \ldots, K$,
- ▷ **Step 2**. *Update configuration*: Set all entries in $X^{n+1} = 0$, except where $k = \arg\min_{k'=1,2,\ldots,K} \|x_i - \mu_{k'}\|^2$, for which $X_{ik}^{n+1} = 1$,
- ▷ **Step 3**. *Update centroids*: Compute $\mu_k^{n+1} = \frac{1}{N_k} \sum_{i=1}^N X_{ik}^{n+1} x_i$ where $N_k = \sum_{i=1}^N X_{ik}^{n+1}$,
- ▷ **Step 4**. *Convergence check*: If $n > 0$ and $X^{n+1} = X^n$, then exit with solution $X^\star = X^{n+1}$ and $\mu^\star = \mu^{n+1}$,
- ▷ **Step 5**. *Iteration*: Update $n = n + 1$ and go back to step 2.

X, so they actually contain the same information.[1] This suggests an idea: start with a good guess for each cluster centroid μ_k, then use these guesses to compute the optimal corresponding partition entries in X_{ik}; now use that to get a better guess for all the μ_k, and so on. The resulting algorithm is known as ***K*-means clustering** (Algorithm (10.1)).

We can show that $F\left(X^{n+1}, \mu^{n+1}\right) \leq F\left(X^n, \mu^n\right)$ the K-means objective (10.3) is *never increasing* (because given μ we can find the X with globally smallest F, and vice-versa). So, the K-means algorithm always converges on a **fixed point** (that is, it finds a minima of F and stays there for all subsequent iterations). But, we cannot know if the minima is local or global, furthermore we cannot know in advance how many iterations it takes to converge (typically 7-15 iterations from experience, but there are no guarantees about this). Another limitation of the algorithm is the potential for *pathological* cases, for instance, in updating configurations (step 2), it is perfectly possible for a cluster to have no data items assigned to it; then it is impossible to find the centroid for that cluster on the next step.[2]

Because of the exponential number of ways of partitioning a set into K clusters, most initial guesses for the configuration X can be substantially improved upon. Figure 10.2 shows the typical evolution of the K-means objective function during a run of the K-means algorithm, showing how the improvement in objective is usually fast at the start and then slows down at convergence. Convergence usually takes only a handful of iterations in practice.

10.3. Example: patch-based digital image dictionaries

Here is an interesting problem in wildlife conservation where a clustering problem arises. Remote, battery-operated digital cameras are set up in the wild to take images of wildlife for ecological purposes such as tracking and counting. Because the cameras are expected to operate entirely remotely for many months at a time, they have to be extremely energy-efficient. This restricts the amount of computational processing they can have on board (fast, capable processors are energy-hungry). All the digital images have to be analyzed to determine whether they should be stored or discarded since memory is limited on the hardware. However, each digital image contains a lot of information. For instance, a single image of around 300 by 600 pixels requires 600kb and the camera take an image

[1] To see this, note that (10.3) is possible to optimize globally, using calculus, with respect to one of either X or μ if the other is fixed. When neither is fixed the problem is intractable, which is of course, the main justification for the K-means algorithm.

[2] Combinatorially, this occurs because we have not excluded partitions which can have empty sets.

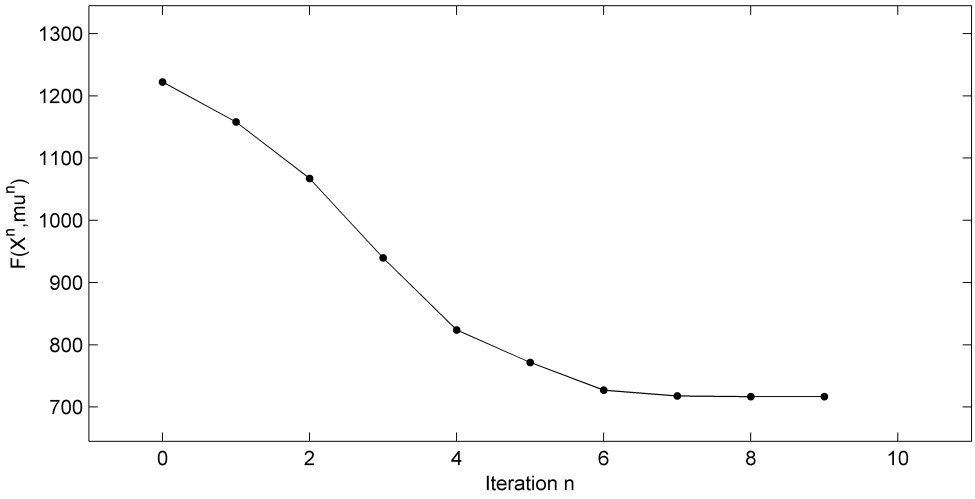

Figure 10.2.: The typical evolution of the K-means objective function value $F\left(X^n, \mu^n\right)$ during iteration of the K-means algorithm (Algorithm 10.1).

every minute, so 1440 images per day means processing close to 1Gb per day. One way to represent these images more efficiently is to simplify them by replacing every small patch of pixels in the image, with a set of patches in a limited **dictionary** of pixel patches. Then, the only information needed to represent an image, is the patches in the dictionary, and for every patch in the image, the associated patch in the dictionary, which can then be used to reconstruct the image. If the dictionary contains sufficiently diverse patches, we expect that not much will be lost by simplifying the images this way.

K-means is applied to the input greyscale (black and white) JPG image of size 328 (vertical) by 640 (horizontal) pixels, with one byte grey scale per pixel (Figure 10.3). This gives an image data size of 209920 bytes. The K-means model has $K = 256$ patches of size $4 \times 4 = 16$ pixels, so that every cluster centroid μ_k is a 4×4 greyscale patch. So, the K-means model has $16 \times 256 = 4096$ bytes. The single byte cluster indices, one per patch in the original image, map patches to clusters. There are $328/4 = 82$ (vertical) by $640/4 = 160$ (horizontal) i.e. 13120 such patches. Therefore, the total amount of space required to store the image in terms of the K-means model is $4096 + 13120 = 17216$ bytes. This represents a ratio of about 12:1 i.e. the image is **compressed**) into only 1/12th of the space of the original. This leads to a substantial saving in computational resources required to process the data.

Image dictionary-based methods such as this perform a similar function to more advanced techniques such as **auto-encoders** used in cutting-edge **vision-language** AI to find patches in images with the same or similar 'meaning' across multiple millions of images, paired with descriptive text. For instance, some of the K-means patches in this image occur much more frequently than the rest, these are associated with visual features such as the sky and grass. On a large enough scale, this kind of statistical association can be used to accurately predict the most likely association of text strings with image patches, for instance, to automatically complete fairly complex queries like "is there more than one wildebeest in this image"?

Figure 10.3.: Using K-means clustering to compress greyscale images for resource sensitive remote sensing applications in conservation. The top panel shows the original image, the bottom panel the image reconstructed from a K-means model with $K = 256$ clusters representing patches of 4×4 pixels.

Learning outcomes for section 10

- ▷ Demonstrate understanding of clustering as an unsupervised machine learning problem: partitioning, indicator functions, centroids, K-means clustering objective.
- ▷ Explain the steps in the K-means clustering algorithm and their significance for optimizing the clustering objective.
- ▷ Apply the K-means clustering algorithm to small data clustering problems.

Section 11.

Classification

Reference reference reading material relevant to this section is **R&N**, Section 18.2, **PRML**, Section 2.5 and **H&T**, Section 13.3.

11.1. Partitioning feature space based on labeled training data

A classification ML model learns to split the given training data into two or more classes, according to the training data. The hypothesis is that there is some **decision (classification) boundary** in the data which makes this classification possible. As with regression, it is **supervised**: given input training data and corresponding labeled output training data, the a training algorithm attempts to find the decision boundary implied by a **classification model** $f(w, x)$. The value of the parameters w determines a specific decision boundary out of all possible boundaries.

As an example of classification, consider the medical problem of creating an ML algorithm to determine whether someone is likely to have a voice pathology or not and if so, pass them on to further medical investigation[1] (Figure 11.1). We are given some training data (x_i, y_i) for $i = 1, 2, \ldots, N$ individuals, which consists of (e.g. in $D = 2$) a pair of *diagnostic feature* data, for instance, two measurable aspects of voice quality, x^1 and x^2, and a suspected diagnosis of pathology ($y = +1$) or not ($y = -1$). Using the classification model $f(w^\star, x)$ where $w^\star$ are a good set of parameters, if this can be made to accurately predict the labels y_i given the x_i as input so $f(w^\star, x_i) \approx y_i$, then we would have confidence that the classification is reliable for test data and possibly for use in medical practice. We do not expect that the classifier is perfect, but we want to make it accurate so as to avoid *clinical errors*. In medical terms, we want the automated classification algorithm to minimize the number of *false positives* (incorrectly triaging someone healthy) and *false negatives* (missing a pathological case).

11.2. The perfect classifier?

Is it possible to construct an ML classifier which never makes any mistakes? Here is a simple idea. Assume that the model $f(w, x)$ acts like a perfect **look-up table** which contains a list of every possible value of the input x and its associated output $y = f(x, w)$. It is easy to see that, for any dataset, this will always make zero classification errors. This seems like a good idea so let us try to make it work for a practical ML example, automatically transcribing handwritten messages. First we need to gather all possible input images of letters. To do this, we first need to plan how much memory we need for the big the look-up table. Take a typical image of $16 \times 16 = 256$ pixels, each of which is greyscale 8 bits. This means $2^{16\times16\times8}$ possible images, which works out at approximately 10^{616} images! Unfortunately, this is completely impractical: even if we could somehow capture images of all possible images, there is no database in the universe large enough to even store them.

[1] In medical terms this kind of activity is called *triage*.

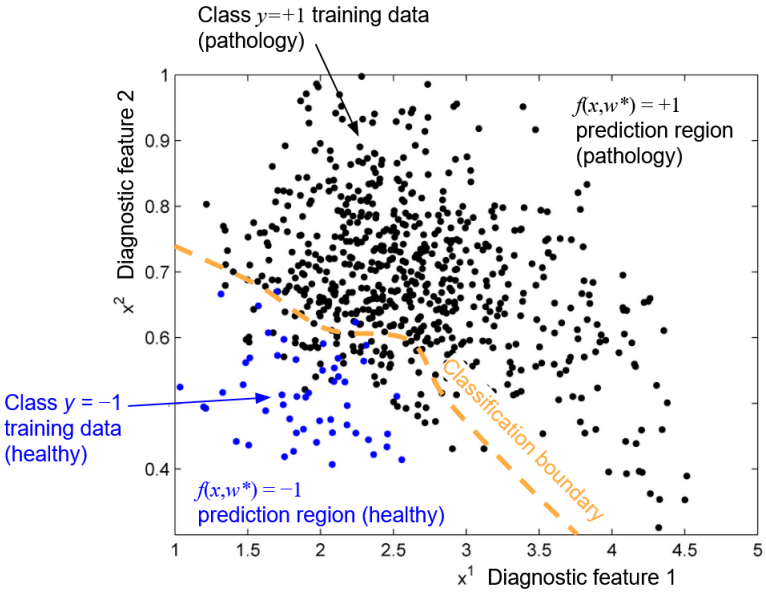

Figure 11.1.: Classification applied to medical triage, illustrating the main concepts in ML classification. A classification model $f(x, w)$ is trained on this the feature data (here, two medical diagnostic features from a set of individuals, x^1, x^2) to find the best parameters $w^\star$ which optimally split the space of the feature data into two regions, one each for class -1 (healthy) and class $+1$ (pathology). The classification (decision) boundary, determined by the classifier $f(x, w^\star)$, splits the feature space into these two regions.

This illustrates a critical point about ML classifiers (and in fact, all supervised models). In principle, any supervised machine learning problem can be completely solved by storing a table of all possible input-output pairs, then prediction on test data is just table look-up. But in practice, all useful ML classifiers are *imperfect models*. The conclusion is that ML is more than just **memorization**, it requires careful (mathematical and computational) **modeling**.

Modern large-scale deep learning classification algorithms trained on extremely large datasets, often behave as if they are merely memorizing the training data, but this cannot literally be true as they are capable of producing outputs for any range of inputs. Complex deep learning algorithms are closer to memorization with some modeling. At the other extreme, **linear classification** models are very simple so they are nearly all modeling and little to no memorization. As with regression, there is usually a trade-off between increasing model complexity and reducing test error. Many complex classifiers can be made to fit the training data with almost exactly zero classification error. But such a classifier may not work very well at all on test data due to the effect of randomness in how the data arises in the real world. Overall, we prefer the simplest model evidenced by the data, but no simpler than that (Figure 11.2).[2]

11.3. Classification defined mathematically

We can state the goal of classification as one of minimizing the **misclassification error** (the sum of incorrectly classified data points in the training data set) of the model $f(w, x)$ on the training data

[2]This does mean that linear models can be (and indeed often are) too simple in practice for most real-world ML applications.

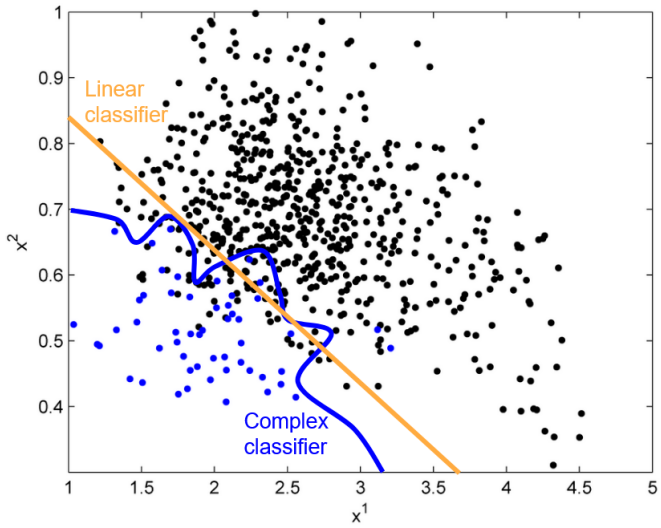

Figure 11.2.: A trade-off usually exists between classifier complexity (reflected in the 'wigglyness of the decision boundary), and prediction accuracy on the test data. A complex classifier can achieve low training error by detailed routing of the decision boundary around the training data, but this is not a good decision boundary if randomness is ignored.

pairs (x_i, y_i),

$$F(w) = \sum_{i=1}^{N} \mathbb{I}\left[f(w, x_i) \neq y_i\right]. \tag{11.1}$$

where $\mathbb{I}[\,]$ is the indicator function given by,[3]

$$\mathbb{I}[P] = \begin{cases} 1 & \text{if logical condition } P \text{ is true} \\ 0 & \text{otherwise} \end{cases} \tag{11.2}$$

This indicator function form of the classification error function, is the reason why (11.1) is also sometimes called the **0-1 loss**. So, the best classifier for some given training data, is the one which minimizes (11.1),

$$w^\star = \arg\min_{w' \in \mathcal{W}} \left(\sum_{i=1}^{N} \mathbb{I}\left[f(w', x_i) \neq y_i\right] \right). \tag{11.3}$$

This optimization problem, while easy to write down, is extremely mathematically challenging to solve. For instance, we might be tempted to use SGD (Algorithm (9.1)) but the the gradient of $1[\,]$ is either *zero* (since the function is flat for nearly every input), or it is *not defined* (in fact, infinite, so we cannot even calculate its value!) In practice then, most classification algorithms do not use the misclassification error function, they use a **proxy or surrogate error (loss)** which is much easier to optimize. A surrogate loss we will study in detail in the next section is the so-called **perceptron loss**,

$$F(w) = \sum_{i=1}^{N} \max\left[0, -y_i f(w, x_i)\right], \tag{11.4}$$

[3]The partition indicator in the clustering section can be considered an example of such a function.

which has much 'nicer' behaviour from a computational perspective. Other surrogate losses, such as the **hinge loss** and **logistic loss** have additional desirable properties and are widely used in practice. Nonetheless, it is important to remember that the problem of optimal classification is only correctly solved by optimizing (11.1) in general, no matter how mathematically convenient a surrogate might be.

Learning outcomes for section 11

- ▷ Explain the principles of supervised classification, including classifier model training, decision boundaries, the contrast between memorization and generalization, 0-1 misclassification error.
- ▷ Demonstrate understanding of the form and importance of surrogate losses, in particular the perceptron loss.

Section 12.

The perceptron

Relevant reference reading material for this section is **PRML**, Section 4.1.7, **R&N**, Section 18.6.3 and **H&T**, Section 4.5.1.

12.1. Linear classification with perceptron loss

The **perceptron** is a simple linear classifier based on a surrogate for the 0-1 loss, the perceptron loss. It is a very important classical algorithm in the history of ML, a direct precursor to modern deep learning algorithms. Because it is extremely simple there are mathematically much better linear algorithms, for instance, **support vector machines**, so the perceptron is rarely used in practice today. Nonetheless, understanding how this classifier works is critical to understanding most of the main principles of modern ML classification.

To see how the perceptron arises, note that we can easily change a linear regression model, $f(w,x) = w^T x$, into a classification model by choosing $f(w,x) = \text{sign}\left(w^T x\right)$. This function is $+1$ if $w^T x > 0$ and -1 if $w^T x < 0$.[1] The decision boundary for this classifier occurs where $w^T x = 0$. For this classifier, the associated misclassification (0-1) error would be $F(w) = \sum_{i=1}^{N} \mathbb{I}\left[\text{sign}\left(w^T x_i\right) \neq y_i\right]$. However, as discussed in previous section, the problem with this 0-1 loss error function is that it is not differentiable, so we cannot find good values of the parameters w by training using gradient descent. Instead, we will use the more convenient **perceptron objective function**,

$$F(w) = \sum_{i=1}^{N} \max\left(0, -y_i w^T x_i\right). \tag{12.1}$$

.

To understand this error function, it helps to focus on the loss expression, $\max\left(0, -y w^T x\right)$ (Table 12.1). There are four different combinations of cases of the sign of the linear model and the true label y; two cases correspond to correct decisions by the model, and two incorrect. When the linear model correctly classifies, the perceptron loss is 0 as with the 0-1 loss. However, if the linear model incorrectly classifies, then the perceptron loss is $w^T x$, which is just the distance to the decision boundary in the direction w. So, the perceptron error 'penalizes' incorrect decisions by the model, to the extent to which they are incorrect, and the perceptron error is the sum of all penalties for the whole data set. This contrasts with the misclassification error, which only counts the number of incorrect decisions in the data set (Figure 12.1).

12.2. SGD for classification: the perceptron algorithm

While the perceptron error does not actually measure the number of misclassified points (because it is a surrogate error), it is differentiable with respect to the linear parameters w. We can therefore apply

[1] The case $w^T x = 0$ leads to $\text{sign}\left(w^T x\right) = 0$ which means x lies exactly on the decision boundary.

Linear regression $w^T x$	**Linear classifier** $\text{sign}\left(w^T x\right)$	**True label** y	$-yw^T x$	**Classifier decision**	**Perceptron loss** $\max\left(0, -yw^T x\right)$	**0-1 loss** $\mathbb{I}\left[\text{sign}\left(w^T x\right) \neq y\right]$
> 0	$+1$	$+1$	$-w^T x$	Correct	0	0
< 0	-1	-1	$w^T x$	Correct	0	0
< 0	-1	$+1$	$-w^T x$	Incorrect	$w^T x$	1
> 0	$+1$	-1	$w^T x$	Incorrect	$w^T x$	1

Table 12.1.: Understanding the perceptron loss function, $\max\left(0, -yw^T x\right)$. This function 'penalizes' incorrect decisions by the linear classifier, by the distance from the decision boundary $w^T x$ in the direction w. This compares to the misclassification (0-1) loss function which assigns the same penalty to all incorrect decisions, regardless of how 'bad' they are.

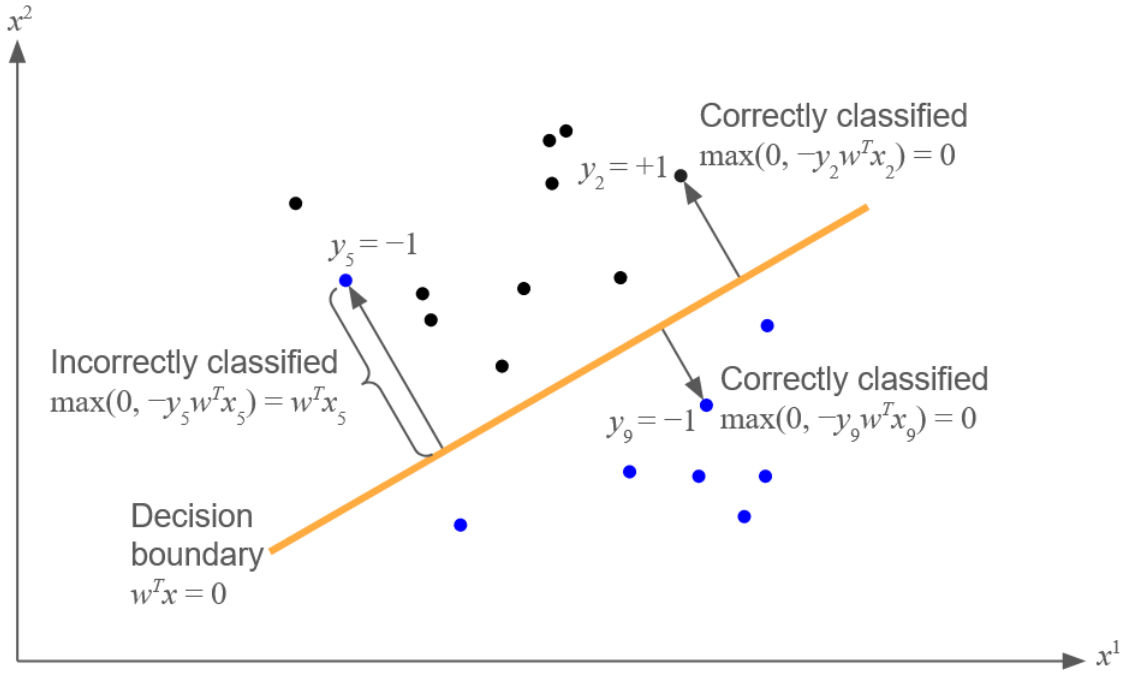

Figure 12.1.: Illustrating the linear perceptron classifier in $D = 2$ feature space with coordinates x^1 and x^2. Data points with label $y = +1$ are black dots, $y = -1$ blue dots. The linear classifier model is $f(w, x) = \text{sign}\left(w^T x\right)$ which implies a decision boundary $w^T x = 0$ (orange line). The perceptron error, $F(w) = \sum_{i=1}^{N} \max\left(0, -y_i w^T x_i\right)$ is the sum of the perpendicular distances (lines with arrowheads) of every misclassified data point to the decision boundary. Correctly classified points contribute 0 to the error (as with the misclassification error).

Algorithm 12.1 The perceptron (training) algorithm.

▷ **Step 1**. *Initialization*: Select a starting candidate classification model w_0, set iteration number $n = 0$, choose maximum number of iterations R and learning rate $\alpha > 0$

▷ **Step 2**. *Gradient descent step*: Compute new model parameters: taking each $i = 1, 2, \ldots, N$ in turn, if sign $\left(w_n^T x_i\right) \neq y_i$, then $w_{n+1} = w_n + \alpha y_i x_i$,

▷ **Step 3**. *Iteration*: If $n < R$, set $n = n + 1$, go back to step 2, else exit with solution $w^\star = w_n$.

SGD (Algorithm 9.1) to training the perceptron classifier. To do this, we need the gradient of (12.1), which is,[2]

$$F_w(w) = -\sum_{i=1}^{N} y_i x_i \mathbb{I}\left[-y_i w^T x_i \geq 0\right]. \tag{12.2}$$

so that the parameter update in the gradient desent step in SGD is,

$$w_{n+1} = w_n + \alpha \sum_{i=1}^{N} y_i x_i \mathbb{I}\left[-y_i w^T x_i \geq 0\right]. \tag{12.3}$$

We can express this in a simple, 'intuitive' form: for all $i = 1, 2, \ldots, N$, if sign $\left(w^T x_i\right) \neq y_i$, then $w_{n+1} = w_n + \alpha y_i x_i$.[3] This is the form we used to describe the **perceptron (training) algorithm** (Algorithm 12.1).

Results of a typical run of the perceptron algorithm given in Table 12.2. This shows that, unlike with SGD applied to linear regression, the perceptron objective function does not always decrease on each iteration: the only situation where the objective function does not increase is where the data is **linearly separable**, that is, it can be perfectly separated into two classes by a linear classifier.

Analysis of this algorithm shows that, it will converge on the perfect boundary (i.e. the one which minimizes the misclassification error) if the data is linearly separable. Otherwise, the algorithm may fail to converge (never settles down on a single solution) and can become **chaotic**. This happens because the algorithm is jumping between different local minima of the objective function. This is the reason why we choose to exit after a fixed number of iterations rather than test for convergence as in the description of the original SGD algorithm.

[2] Can be derived using basic properties of calculus but we postpone this until the section on automatic differentiation.

[3] To see why this arises, note that $1\left[-y_i w^T x_i \geq 0\right] = 1\left[\text{sign}\left(w^T x_i\right) \neq y_i\right]$ and then expand the indicator function $1[\,]$.

Iteration	**Parameters** w_n	**Error function value** $F(w_n)$	**Error function gradient** $F_w(w_n)$
$n = 0$ (initialize)	$[-0.94, 0.34]$	4.59	$[-5.63, -2.14]$
$n = 1$	$[-0.66, 0.44]$	2.77	$[-5.63, -2.14]$
$n = 2$	$[-0.38, 0.55]$	0.96	$[-5.63, -2.14]$
$n = 3$	$[-0.10, 0.66]$	0.56	$[-5.63, -2.14]$
$n = 4$	$[-0.18, 0.60]$	0.43	$[1.60, 1.09]$
$n = 5$	$[-0.14, 0.58]$	0.41	$[-0.66, 0.53]$
$n = 6$	$[-0.20, 0.53]$	0.44	$[1.10, 0.99]$
$n = 7$	$[-0.12, 0.51]$	0.38	$[-1.54, 0.25]$
$n = 8$	$[-0.18, 0.46]$	0.39	$[1.10, 0.99]$
$n = 9$ (exit)	$[-0.10, 0.45]$	0.34	$[-1.54, 0.25]$

Table 12.2.: Example output from the perceptron algorithm (Algorithm 12.1) for data of dimension $D = 2$, with input data size $N = 20$, data shown in Figure 12.2. This data is not linearly separable. Algorithm parameters: learning rate $\alpha = 0.05$ and maximum number of iterations, $R = 10$. In this run, the algorithm terminates with result $w^\star = [-0.10, 0.45]$ and $F(w^\star) = 0.34$.

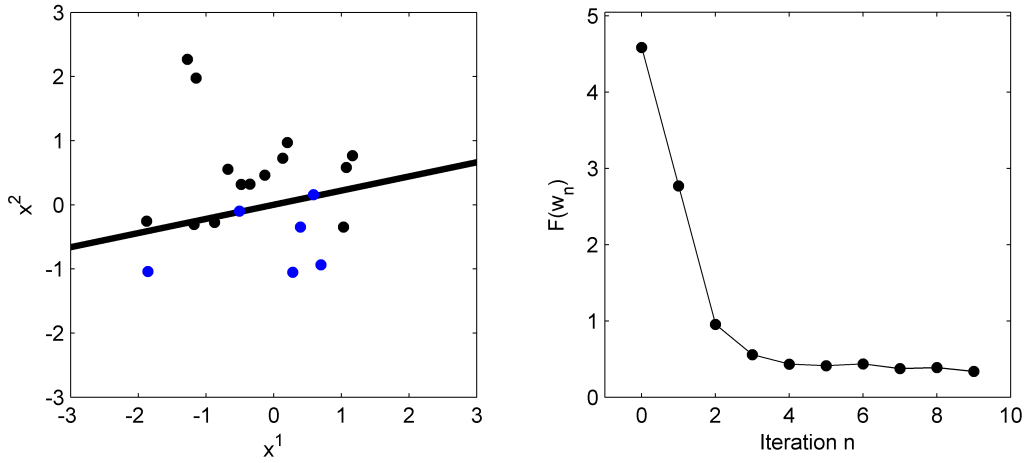

Figure 12.2.: Example evolution of the perceptron algorithm (Algorithm 12.1) for data of dimension $D = 2$, with input data size $N = 20$. This data is not linearly separable. Algorithm parameters: learning rate $\alpha = 0.05$ and maximum number of iterations, $R = 10$. In this run, the algorithm terminates with result $w^\star = [-0.10, 0.45]$ and $F(w^\star) = 0.34$.

Learning outcomes for section 12

- ▷ Demonstrate understanding of the perceptron classifier: linear classification modelling, function and geometric meaning of the perceptron objective function.
- ▷ Have knowledge of and be able to explain the steps in the perceptron training algorithm.
- ▷ Apply the perceptron training algorithm to small classification problems.

Section 13.

Neural networks and deep learning

Relevent reference reading material for this section: **PRML**, Section 5.1 and **H&T**, Section 11.3.

13.1. Activation nonlinearities

So far, we have only encountered the ReLU activation function arising in the perceptron. Nonlinear functions used in deep networks can, in principle be any nonlinear function, but to be useful in practice they ought to be easy to use in calculations, for instance, to be easy to compute and simple to differentiate (Table 13.1 and Figure 13.1). It is a notable feature of activation functions, that often either the function or its gradient is used. By far the most widely used activation function it the ReLU function, probably because of its simplicity: for half of its range it is exactly zero which simplifies computations in complex deep nets substantially.

13.2. Deep neural networks: chained perceptrons

We can view the simple linear perceptron model $f(w, x) = \max\left(0, w^T x\right)$ in the form of a **weighted linear combination** with **nonlinear activation function**,

$$y = \max\left(0, w^T x\right). \tag{13.1}$$

This interpretation is inspired by the biological neuron, which has **axons**, **dendrites** and a **nucleus**, which are organized so as to 'fire' (produce a non-zero output) when the sum of the inputs crosses a certain threshold level.[1] The perceptron is very limited and can only model linear decision boundaries. In practical ML applications, more complex nonlinear boundaries are required. To extend the capability of these simple models, we can chain them together so the output of one, feeds into the input of another. For instance, a set of neurons acting on the same inputs x, whose outputs z are then

[1] This is only a very vague and inaccurate description of how neurons actually work!

Activation function	Expression	Derivative	Expression
ReLU (rectified linear unit)	$\max(0, x)$	Step function	$\mathbb{I}[x \geq 0]$
Softplus	$\ln(1 + e^x)$	Logistic (sigmoid)	$\frac{1}{1+e^{-x}}$
Hyperbolic tangent	$\tanh(x)$	Hyperbolic tangent gradient	$1 - \tanh(x)^2$

Table 13.1.: A selection of widely-used nonlinear activation functions in modern deep neural networks, and their corresponding gradient functions.

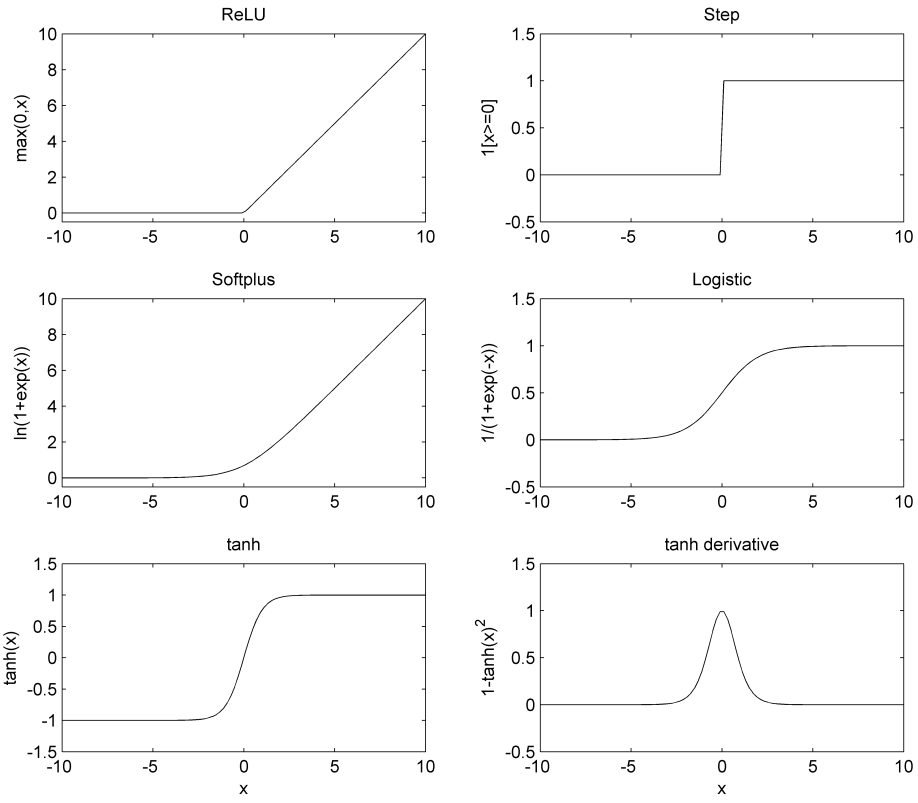

Figure 13.1.: Widely encountered nonlinear activation functions used in modern deep neural networks. The column on the right are all derivatives of the corresponding function in the same row on the left.

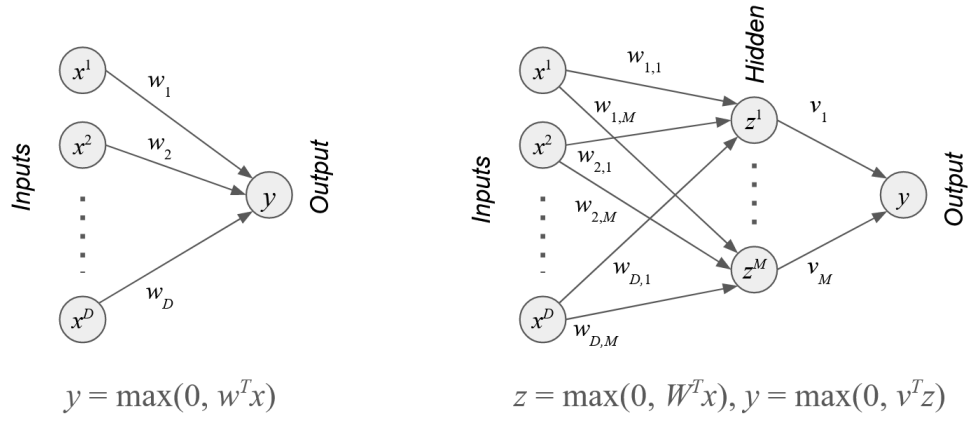

Figure 13.2.: (Left) The perceptron as a single *neuron* which takes multiple inputs, sums them together and applies a nonlinear activation (here, the ReLU) function to create the output, $y = \max\left(0, w^T x\right)$ (Right) Extending the perceptron to two layers, with nonlinear activations connecting them through *hidden* neurons z whose output is fed into the *output layer* creating the final output y, e.g. $z = \max\left(0, W^T x\right)$ and $y = \max\left(0, v^T z\right)$, leads to the basic multi-layer perceptron (MLP). This is the simplest (fully-connected, two-layer) deep learning algorithm.

fed into the input of another neuron, leads to the simple two-layer **multilayer perceptron** (MLP):

$$\begin{aligned} z &= \max\left(0, W^T x\right) \\ y &= \max\left(0, v^T z\right), \end{aligned} \tag{13.2}$$

where W is a **matrix** of weights, so that each column is a vector of weights for each of the M hidden perceptrons in the second layer,

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,M} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D,1} & w_{D,2} & \cdots & w_{D,M} \end{bmatrix}, \tag{13.3}$$

and $v_1, v_2, \ldots, v_M$ are the weights for the perceptron at the output layer (Figure 13.2).

Extending to multiple layers in this way, with more than two sets of hidden neurons, leads to the number of total number weights growing rapidly (for example, for three fully connected layers with D inputs, M nodes in the first hidden layer, N in the second layer and a single output node, the number of weights is $DM + MN + N$). Furthermore, there are situations where it makes sense to have the weights **shared** between connections. A classical example of this is the **convolutional neural network (CNN)** which is well-suited to ordered data such as images and time series (Figure 13.3). For instance, in image data, it does not usually matter where exactly in the image an object is located – it could be translated up/down or left/right[2] – so an object detection neural network might have weight sharing in overlapping patches (windows).

[2]This property is known as *translation invariance*.

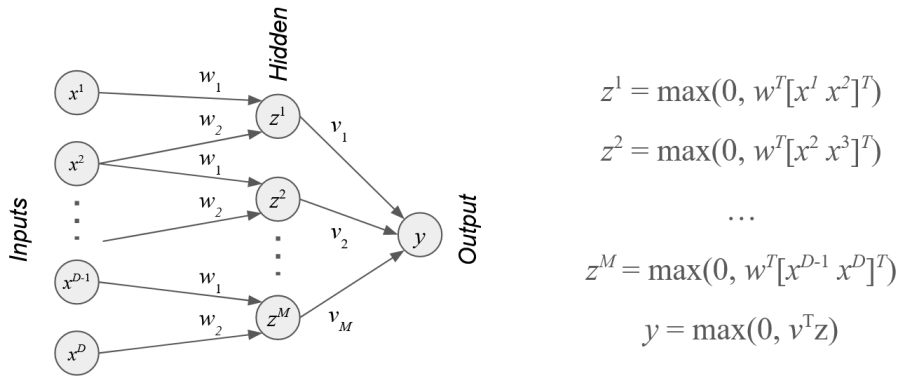

Figure 13.3.: A simple convolutional neural network (CNN), which shares two weights in the input layer across the whole input sequence, e.g. $w_1x^1 + w_2x^2$, $w_1x^2 + w_2x^3$, $w_1x^3 + w_2x^4$ etc. This allows the network to learn patterns where the specific location in the sequence of the input, does not matter, and is well suited to time series (it is said to have 'time shift invariance').

13.3. Example: deep logic networks

Multilayer deep networks can compute pretty much any desired function of the inputs. Here we will explore the use of deep networks for a problem in symbolic AI: computing **logical functions** such as the connectives in propositional calculus (Section 8).

To illustrate the idea, the activation we will use is the function $\text{sign}(x)$ which is $+1$ if $x > 0$, -1 if $x < 0$ and 0 otherwise. This is much like the step function in Figure 13.1. The goal will be to use the output of the activation $\text{sign}(x)$ with -1 representing $False$, and $+1$ representing $True$. We will construct a system of logical computation based on the use of the neural network function $f_b(w, x) = \text{sign}(w_0 + w_1x^1 + w_2x^2)$[3] for the binary operators 'and' and 'or', and for the 'not' operator we will use the single input neural network function $f_u(w, x) = \text{sign}(w_0 + w_1x^1)$. For the 'and' function, under this encoding, $w_{\text{and}} = [-1, 1, 1]$ behaves as required. Similarly, for the 'or' function, weights $w_{\text{or}} = [1, 1, 1]$ work, and for the 'not' function, $w_{\text{not}} = [0, -1]$ suffices.[4] So, our single-layer logical operator neurons are given by the following very simple functions,

$$\begin{aligned} f_{\text{and}}\left(x^1, x^2\right) &= \text{sign}\left(x^1 + x^2 - 1\right) \\ f_{\text{or}}\left(x^1, x^2\right) &= \text{sign}\left(x^1 + x^2 + 1\right) \\ f_{\text{not}}(x) &= \text{sign}(-x)\,. \end{aligned} \tag{13.4}$$

as is straightforward to check, with reference to the logical truth tables for these operators. We can visualize these as single layer linear neurons (Figure 13.4).

Given these functions, we can now construct any desired logical function, by translating the corresponding logical expression into the corresponding neural network function, which will be an appropriate composite of the three functions (13.4). For instance, the 'xnor' function, which is true if the value of the two inputs agree, is given by the logical expression $y(u, v) = (u \wedge v) \vee (\neg u \wedge \neg v)$, which

[3] This is in the familiar form $w^T x$ if $x^1 = 1$ and we index dimensions starting at 0 rather than 1.

[4] These weights were determined by solving a system of (*overdetermined*) *linear equations* for each operator. In fact, this can be done for any reasonable activation function.

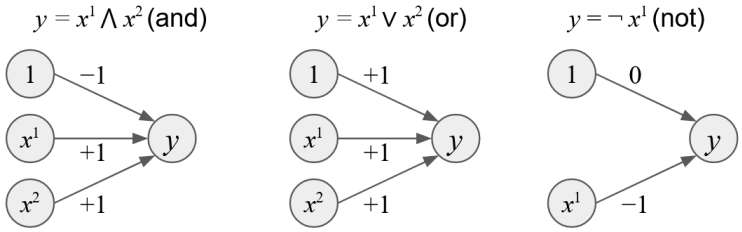

Figure 13.4.: Implementing the three fundamental logical connectives, 'and', 'or' or 'not' using single-layer linear neurons with sign activation function, using the encoding $+1$ for $True$ and -1 for $False$..

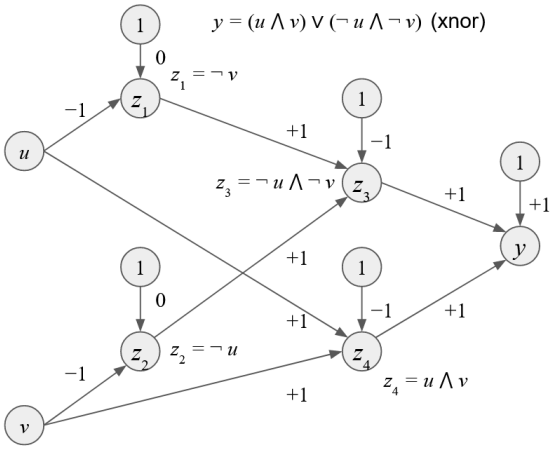

Figure 13.5.: The 'xnor' function $y(u, v) = (u \wedge v) \vee (\neg u \wedge \neg v)$ implemented as a multilayer neural network with two hidden layers, using neurons with sign activation function. The logical encoding is $+1$ for $True$ and -1 for $False$. (The constant nodes are essentially not inputs so they have been moved out of the way of the inputs for clarity).

in terms of the neural network functions is,[5]

$$y(u, v) = f_{\text{or}}\left(f_{\text{and}}(u, v), f_{\text{and}}\left(f_{\text{not}}(u), f_{\text{not}}(v)\right)\right). \tag{13.5}$$

When visualized, we can see that (in this implementation at least) we need two hidden layers (Figure 13.5), the first hidden layer to compute the intermediate terms $\neg u$, $\neg v$ and $u \wedge v$, and the second to compute the intermediate result $\neg u \wedge \neg v$.[6] In fact, the 'xnor' function is an example of a simple logical function which a single layer linear neuron cannot compute, which shows the need for multilayer networks.

[5] This is the *disjunctive normal form* for the 'xnor' function.

[6] There is a way to simplify this to remove one hidden layer if we use the fact that $\text{sign}(x) = x$, provided the inputs are always logical values -1 and $+1$.

Learning outcomes for section 13

- ▷ Demonstrate understanding of the fundamental principles of deep learning: activation nonlinearities, linear transformations, chained perceptrons, fully connected layers.
- ▷ Explain the significance and mathematical structure of weight sharing in convolutional networks.
- ▷ Perform computations using simple deep networks.

Section 14.

Automatic differentiation

Relevant reference reading material for this section: **PRML**, Section 5.3, **R&N**, Section 18.7 and **H&T**, Section 11.4 and Section 11.7.

14.1. Deep neural networks and automatic differentiation

Gradient descent is an extremely versatile method for producing good solutions to optimization problems in ML. Single layer linear perceptron training was shown to be relatively straightforward: just use SGD to minimize the perceptron objective $F(w)$ with respect to the weights w, given the input training input data x and labels y. The expression (12.2) for the gradient $F_w(w)$ is not too complicated to calculate by hand. However, to train hidden layer weights v in deep networks, we must propagate gradients from the output error $F(w)$, all the way through each layer. Analytical gradient expressions quickly become intractable. Noting that we do not generally need to care about the specific form of this gradient function, only that it can be easily calculated, motivates the use of more sophisticated gradient calculation methods. Widely used are **backpropagation** and **automatic differentiation (AD)** There are many different backpropagation and AD schemes, principally **forward** and **reverse mode**; depending upon whether the gradient computations start with values at the input to the network, or start with values obtained at the output instead. We will discuss forward mode AD for its conceptual simplicity.[1]

Automatic differentiation is a **meta-programming** approach to gradient calculation. By this it is meant: the actual primitive computations which are carried out by a computer to evaluate some ML function, are 'shadowed' by simultaneous calculation of the gradient of that function in the background (Figure 14.1).[2] Unlike **numerical gradients**, AD computations are exact in the sense that, if the gradient functions of the primitives are accurate, then the final gradients will generally be accurate too. Unlike **(analytical) symbolic differentiation**, the full mathematical expression for the gradient is rarely, if ever, actually calculated; the goal is to efficiently compute accurate gradients the point evaluated by the function that is to be differentiated, rather than anything other aims.

Software packages such as **PyTorch** and **JAX** help the programmer largely avoid the need for finding any hand-computed gradients in this way and have become an essential tool in deep learning research and AI industries. In this section we will study how the technique works and carry out some toy calculations to demonstrate this; in practice, gradient computations involve substantial computations and are therefore best explored computationally.

[1] There are different computational trade-offs between AD schemes, even though all schemes should, in principal, compute the exact same value of the gradient.

[2] Some ML researchers call AD *differentiable computing* for this reason.

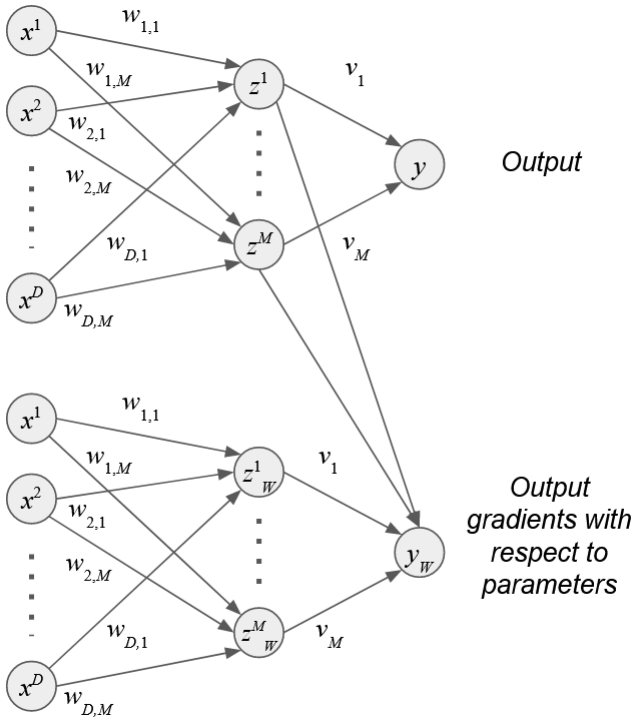

Figure 14.1.: Automatic differentiation (AD) obtains the gradients of the output of a deep neural network simultaneous with the output of the network itself, using the information available to, or computed by, the network at each stage.

14.2. Chain rule of calculus and dual numbers

Nearly all functions computing anything signficantly useful in ML, involve **composition**, that is, they require chains of functions applied to the output of other functions. Gradients of composite functions such as $F(f(x))$, are found using the **chain rule of calculus**:

$$\frac{d}{dx}F(f(x)) = F'(f(x)) \times f'(x), \tag{14.1}$$

where $f'(u)$ is shorthand for $\frac{df}{du}(u)$. As can be seen, this involves the gradients of the first function in the composite, $f'(x)$ and the second function $F'(u)$, along with the value of the first function, $f(x)$. The output of the composite $F(f(x))$, will generally become the first function in the next composite, and so on. Therefore, to calculate the gradient of a composite function, the value of the first function is also needed, which implies that for computational efficiency reasons it makes sense to compute the gradient along with the value of the function at the same time. **Dual numbers**, represented as a pair (u, u'), are a convenient algebraic tool to handle such chained computations, which keeps track of both the current computation's value u and the value of its' derivative u' together, as a sequence of functions, f, are applied,[3]

$$f((u, u')) = (f(u), f'(u)u'), \tag{14.2}$$

[3]For those familiar with *complex numbers*, dual numbers are similar in that the real line is extended to the two-dimensional plane with the additional *infinitessimal* coordinate, ϵ, along with the rule that $\epsilon^2 = 0$, much as $i^2 = -1$ in complex numbers.

As long as the gradient of the current transformation f in the chain is available analytically, it is possible to compute the composite of any number of functions in this way. Indeed, most forms of computational operations used in ML are special cases of this rule. Here are some which are widely encountered in, for example, deep neural networks:[4]

▷ **Addition** $f(u,v) = u+v$,

$$(u,u') + (v,v') = (u+v, u'+v'). \tag{14.3}$$

▷ **Multiplication** $f(u,v) = uv$,

$$(u,u') \times (v,v') = (uv, u'v + v'u). \tag{14.4}$$

▷ **Maximum** $f(u,v) = \max(u,v)$,

$$\max((u,u'),(v,v')) = (\max(u,v), u'\mathbb{I}[u>v] + v'\mathbb{I}[u\leq v]). \tag{14.5}$$

▷ **ReLU activation** nonlinearity $f(u) = \text{relu}(u)$,

$$\text{relu}((u,u')) = (\max(0,u), u'\mathbb{I}[u\geq 0]). \tag{14.6}$$

▷ **Constants** $f(u) = c$,

$$f((u,u')) = (c,0),$$

reflecting the fact that this function is (by definition of being constant) independent of the differentiation variable and therefore has zero gradient.

▷ **Variables** $f(u) = u$,

$$f((u,u')) = (u,1),$$

such as variable parameters in a neural network, since by definition $\frac{du}{du} = 1$.

Here is an example of a chained calculation carried out using dual numbers. Given the constants $y=3$ and $z=-1$ and variable $x=2$, compute $u(x,y,z) = \max(yz, y+2x)$ and its derivative, $u_x(x,y,z)$. Applying the rules above successively (and using additional symbols for intermediate computational results),

$$\begin{aligned}
\bar{x} &= (2,1)\\
\bar{y} &= (3,0)\\
\bar{z} &= (-1,0)\\
c &= (2,0)\\
c\bar{x} &= (2,0)\times(2,1) = (4,2)\\
r_1 &= \bar{y}\times\bar{z} = (3,0)\times(-1,0) = (-3,0)\\
r_2 &= \bar{y} + c\bar{x} = (3,0)+(4,2) = (7,2)\\
\bar{u} &= \max((-3,0),(7,2)) = (7,2),
\end{aligned} \tag{14.7}$$

therefore $u(x,y,z) = 7$ and $u_x(x,y,z) = 2$. While it is, of course, always possible to find the symbolic derivative of the function $u(x,y,z)$, AD enables entirely 'mechanical' calculational steps which lends itself to software implementation.

[4] Some of these are just familiar rules from calculus, such as the sum and product rules.

	Dual number values $\left(u, u' = \frac{du}{dw_{1,1}}\right)$
Inputs	$\bar{x}_1 = (1.89, 0)$, $\bar{x}_2 = (-2.94, 0)$
Linear layer weights	$\bar{w}_{1,1} = (0.98, 1)$, $\bar{w}_{2,1} = (-0.27, 0)$, $\bar{w}_{1,2} = (-0.55, 0)$, $\bar{w}_{2,2} = (-0.10, 0)$
Hidden node 1 input	$\bar{w}_{1,1}\bar{x}^1 + \bar{w}_{2,1}\bar{x}^2 = (0.98, 1) \times (1.89, 0) + (-0.27, 0) \times (-2.94, 0) = (2.63, 1.89)$
Hidden node 2 input	$\bar{w}_{1,2}\bar{x}^1 + \bar{w}_{2,2}\bar{x}^2 = (-0.55, 0) \times (1.89, 0) + (-0.10, 0) \times (-2.94, 0) = (-0.75, 0)$
Hidden node 1 output	$\bar{z}^1 = \max\left((0, 0), (2.63, 1.89)\right) = (2.63, 1.89)$
Hidden node 2 output	$\bar{z}^2 = \max\left((0, 0), (-0.75, 0)\right) = (0, 0)$
Linear layer weights	$\bar{v}_1 = (-1.38, 0)$, $\bar{v}_2 = (-0.73, 0)$
Output node input	$\bar{v}_1\bar{z}^1 + \bar{v}_2\bar{z}^2 = (-1.38, 0) \times (2.63, 1.89) + (-0.73, 0) \times (0, 0) = (-3.63, -2.60)$
Label	$\bar{y} = (1, 0)$
Output	$F(w_{1,1}) = \max\left((0, 0), (-1, 0) \times (1, 0) \times (-3.63, -2.60)\right) = (3.63, 2.60)$

Table 14.1.: Example automatic differentiation (AD) computations using dual numbers, of the output of a two-layer multilayer perceptron with ReLU activation and perceptron loss.

14.3. Example: AD for multilayer perceptrons

AD applied to deep learning is straightforward. For example, the simple two layer network with two hidden nodes and $D = 2$ dimensional input with ReLU activations and perceptron loss, has the forward computation,

$$\begin{aligned} z^1 &= \text{relu}\left(w_{1,1}x^1 + w_{2,1}x^2\right) \\ z^2 &= \text{relu}\left(w_{1,2}x^1 + w_{2,2}x^2\right) \\ F(w_{1,1}) &= \text{relu}\left(-y \times \left(v_1 z^1 + v_2 z^2\right)\right). \end{aligned} \tag{14.8}$$

The resulting computations are shown in Table 14.1. Gradients of the output with respect to the rest of the parameters, e.g. $F(w_{1,2})$ and $F(v_1)$ etc. are computed similarly; in practice all gradients are computed in parallel using vector arithmetic.

Learning outcomes for section 14

- ▷ Be able to explain the purpose of automatic differentiation, its importance for deep learning, and how it differs from numerical and symbolic differentation.
- ▷ Explain and demonstrate how automatic differentiation can be computed using dual numbers.
- ▷ Perform simple gradient computations using dual numbers.

Section 15.

Probability and probabilistic graphical models

Relevant reference reading material for this section are **MLSP**, Section 1.4, Section 1.6, Section 5.1, Section 5.2, **PRML**, Section 8.1, Section 8.2 and **R&N**, Section 14.1, Section 14.2.

15.1. Rules of probability and probability distributions

The AI and ML algorithms discussed this far in the course are not statistically precise, but, particularly in scientific or engineering applications, we need a quantifiable treatment of uncertainty, and this requires **probability**. Probability provides a mathematical calculus for reasoning about **random events**, conveniently indexed using **random variables (RV)** which can be **discrete** or **continuous**. **Probability functions** obey the rules (**axioms**) **of probability**, which are:

1. $\Pr(A) \geq 0$,
2. $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ provided $A \cap B = \emptyset$,
3. $\Pr(\Omega) = 1$.

Here, $\Pr(A)$ refers to the probability of random event A, Ω refers to the set of **all possible events**[1] which is also known as the **sample space** of the model, and $\emptyset$ is the **empty set** (the impossible event). In words, the first rule states that probabilities are non-negative real numbers. The second states that the probabillity of non-overlapping events is the sum of their individual probabilities. The third states that the probability of any event occurring, is 1, i.e. it is inevitable that some event occurs.

Probability models come in two main kinds: discrete and continuous. Discrete **probability mass functions (PMF)** give the probability that the RV X takes on the value x, written in shorthand as $P(X = x)$. It is an actual probability so it lies in the range$[0, 1]$. Furthermore it must be normalized, so $\sum_{x \in \Omega_X} P(X = x) = 1$, where Ω_X is the sample space for X. A continuous **probability density function (PDF)** $p(x)$ gives the amount of **probability per unit** (**probability density**). A PDF must satisfy $p(x) \geq 0$ for all $x \in \Omega_X$. The volume under this function (a generalization of the area under a unidimensional PDF), gives the probability of the event represented by that volume and is calculated using integration, $\int_{x \in A} p(x)\, dx = \Pr(A)$ and it must be normalized, $\int_{\Omega_X} p(x)\, dx = 1$.[2] For the purposes of this course "summation for discrete, integration for continuous" is a good heuristic under which nearly all of probability calculus is the same for both discrete and continuous distributions. Some examples of discrete RVs follow.

[1] As a technical point of detail, events can be combinations of outcomes, including single outcomes, which are elements of the *sigma-algebra* on the sample space.

[2] Note that this does *not* mean that $p(x) \leq 1$!

The **Bernoulli distribution**, which is a **binary** RV, is a good model for a 'biased coin flip'. With the sample space $\Omega_X = \{0, 1\}$, the associated probability (mass) function is given by $P(X = 0) = 1-p$ and $P(X = 1) = p$, where $p \in [0, 1]$ is the only parameter. This is normalized because,

$$\begin{aligned} \sum_{x \in \{0,1\}} P(X = x) &= P(X = 0) + P(X = 1) \\ &= 1 - p + p \\ &= 1. \end{aligned} \tag{15.1}$$

The case $p = \frac{1}{2}$ represents a fair coin.

The (discrete) **uniform distribution** has a finit

e sample space such as $\Omega_X = \{1, 2, \ldots, N\}$, where each event is equally likely, $P(X = x) = \frac{1}{N}$, so it is normalized, $\sum_{x=1}^{N} P(X = x) = \sum_{x=1}^{N} \frac{1}{N} = 1$. An obvious example is the fair dice with $N = 6$, so $P(X = x) = \frac{1}{6}$.

The uniform distribution is a special case of the **categorical distribution** where each outcome can have a different probability, so $P(X = x) = p_x$, which means there are N parameter values in the length-N vector, indexed by x. Each p_x is a probability value, so it must satisfy $0 \leq p_x \leq 1$. For example, for $N = 3$, the categorical distribution $p = [0.25, 0.25, 0.5]$ has $p_1 = 0.25$, $p_2 = 0.25$ and $p_3 = 0.5$.

A more complex distribution is the **binomial distribution** which can be considered a model of a sequence of independent, but biased, coin flips (i.e. Bernoulli distributed events). It can be shown that the probability of any sequence of flips with $x \leq N$ heads is $(1 - p)^{N-x} p^x$. The number of ways of obtaining x heads in this sequence is given by the **binomial coefficient**,

$$\binom{N}{x} = \frac{N!}{x!\,(N - x)!} \tag{15.2}$$

This is the number of combinations of k elements among N, from which we obtain the binomial mass function,

$$P(X = x) = \binom{N}{x} (1 - p)^{N-x} p^x. \tag{15.3}$$

Perhaps the most ubiquitous continuous distribution, is the **Gaussian** or **normal distribution**. In the $D = 1$ case, this has the density function,[3]

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

on the sample space $\Omega_X = \mathbb{R}$, the whole of the real line. The parameter $\mu \in \mathbb{R}$ is the **mean** or **average** of the distribution, which locates the highest probablity density value of the function, and $\sigma^2 > 0$ is the **variance** which measures the 'width' or 'spread' of the density around the mean (Figure 15.1).

The Gaussian has many remarkable properties. Among these: the mean, median (value which has as much probability on one side as the other) and mode (the highest probability value) are all equal to μ, and the distribution of a sum of an infinite number of random variables (with finite mean and variance), is Gaussian (via the famous **central limit theorem**). For this reason (and many others), the Gaussian is widely used in probabilstic ML and AI.

[3] This normal distribution is a special case of the *multivariate Gaussian* for $D > 1$.

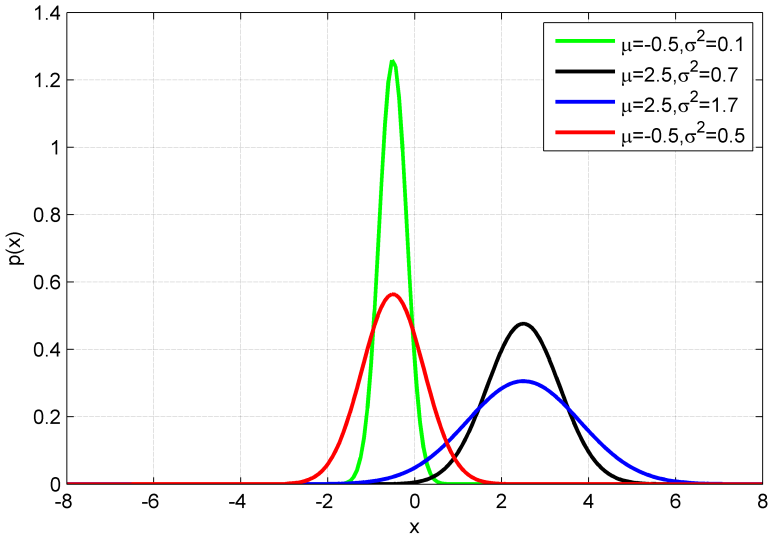

Figure 15.1.: Density function (PDF) of the Gaussian distribution for a variety of mean μ and variance σ^2 parameter values.

15.2. Two or more random variables together: marginals, conditionals, independence

Two or more events occurring simultaneously are represented by **joint RVs** and corresponding joint PMFs and/or PDFs. For the case of two discrete RVs, X and Y, the joint mass function is,

$$\Pr\left(X = x \text{ and } Y = y\right) = P\left(X = x, Y = y\right), \tag{15.4}$$

defined on the joint sample space $\Omega_X \times \Omega_Y$. As an example, consider the case of a coin flip paired with a dice throw, the joint sample space has $2 \times 6 = 12$ possible outcomes,

$$\Omega_X \times \Omega_Y = \{(H,1), (H,2), (H,3), (H,4), (H,5), (H,6), (T,1), (T,2), (T,3), (T,4), (T,5), (T,6)\}$$

As a proper distribution function, the joint distribution must be normalized, so $\sum_{(x,y)\in\Omega_X\times\Omega_Y} P\left(X = x, Y = y\right) = 1$. For two continuous RVs, the joint PDF $p\left(x, y\right)$ is the density value at $x \in \Omega_X$ and $y \in \Omega_Y$ simultaneously. Clearly, joint density functions should be normalized, so we must have $\int_{\Omega_X\times\Omega_Y} p\left(x, y\right) dx\, dy = 1$. Since the laws of probability apply equally to the discrete and continuous cases, we will usually use the shorthand $P_{XY}\left(X, Y\right)$ for the joint distribution function of the random variables X and Y, or where there is no ambiguity just $P\left(X, Y\right)$.

A joint distribution contains all information about the joint RVs, so for instance, we can find the distribution function of one of the RVs from the joint, by **marginalization**,

$$\begin{aligned} P\left(Y = y\right) &= \sum_{x\in\Omega_X} P\left(X = x, Y = y\right) \\ P\left(X = x\right) &= \sum_{y\in\Omega_Y} P\left(X = x, Y = y\right), \end{aligned} \tag{15.5}$$

and the same result applies to continuous distributions using integration rather than summation.

Joint $P(X=x, Y=y)$	$y=0$	$y=1$
$x=0$	$\frac{3}{7}$	$\frac{1}{7}$
$x=1$	$\frac{3}{15}$	$\frac{8}{35}$

Marginal $P(X=x)$	
$x=0$	$P(X=0,Y=0)+P(X=0,Y=1)=\frac{4}{7}$
$x=1$	$P(X=1,Y=0)+P(X=1,Y=1)=\frac{3}{7}$

Marginal $P(Y=y)$	
$y=0$	$P(X=0,Y=0)+P(X=1,Y=0)=\frac{22}{35}$
$y=1$	$P(X=0,Y=1)+P(X=1,Y=1)=\frac{13}{35}$

Conditional $P(X=x\|Y=y)$	$y=0$	$y=1$
$x=0$	$\frac{P(X=0,Y=0)}{P(Y=0)}=\frac{15}{22}$	$\frac{P(X=0,Y=1)}{P(Y=1)}=\frac{5}{13}$
$x=1$	$\frac{P(X=1,Y=0)}{P(Y=0)}=\frac{7}{22}$	$\frac{P(X=1,Y=1)}{P(Y=1)}=\frac{8}{13}$

Conditional $P(Y=y\|X=x)$	$y=0$	$y=1$
$x=0$	$\frac{P(X=0,Y=0)}{P(X=0)}=\frac{3}{4}$	$\frac{P(X=0,Y=1)}{P(X=0)}=\frac{1}{4}$
$x=1$	$\frac{P(X=1,Y=0)}{P(X=1)}=\frac{7}{15}$	$\frac{P(X=1,Y=1)}{P(X=1)}=\frac{8}{15}$

Table 15.1.: Illustrating all marginal and conditional distributions for a joint categorical distribution over the joint sample space $\Omega_X \times \Omega_Y = \{0,1\} \times \{0,1\}$.

Joint $P(X=x, Y=y)$	$y=0$	$y=1$
$x=0$	0.30	0.02
$x=1$	0.08	0.60

Marginal $P(X=x)$	
$x=0$	0.32
$x=1$	0.68

Marginal $P(Y=y)$	
$y=0$	0.38
$y=1$	0.62

Factored joint $P(X=x)\,P(Y=y)$	$y=0$	$y=1$
$x=0$	$P(X=0)\,P(Y=0)=0.122$	$P(X=0)\,P(Y=1)=0.198$
$x=1$	$P(X=1)\,P(Y=0)=0.258$	$P(X=1)\,P(Y=1)=0.422$

Table 15.2.: Illustrating independence between random variables, over the joint categorical sample space $\Omega_X \times \Omega_Y = \{0,1\} \times \{0,1\}$. The fact that $P(X,Y) \neq P(X)\,P(Y)$ means that X and Y are not independent.

Properly normalized, fixing one variable allows the joint to act as a new distribution called the **conditional**,

$$\begin{aligned} P(X=x|Y=y) &= \frac{P(X=x,Y=y)}{P(Y=y)} \\ P(Y=y|X=x) &= \frac{P(X=x,Y=y)}{P(X=x)}, \end{aligned} \tag{15.6}$$

and we use the shorthand $P(X|Y)$ and $P(Y|X)$. Both marginals and conditionals can be extended to multiple joint random variables through summation or division by joint subsets. See Table 15.1 for a worked example in the case of the joint categorical distribution.

If the probability of a conditional distribution $P(X|Y)$ is unaffected by the conditioning variable Y, we say that X is **independent** of Y. In other words, if $P(X|Y) = P(X)$, then we can rearrange this to write $P(X,Y) = P(X)\,P(Y)$ so the joint distribution is factored into a product of the marginal distributions (Table 15.2).

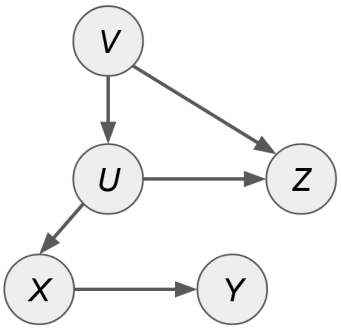

Figure 15.2.: Simple example probabilistic graphical model over the five RVs V, U, X, Y and Z. We can immediately see the child/parent relationships between variables; some of them: V is a parent of both U and Z, Y is a child of X and X is a child of U in turn. We can therefore read off the conditional independence relationships, for example, $P(Z|V,U) = P(Z|V,U,X,Y)$ (in words, Z is conditionally independent of X and Y given U and V), $P(Y|X) = P(Y|X,Z)$ (Y is conditionally independent of Z given X).

Independence/conditional independence such as this, is of critical important in probabilistic AI and ML, since it allows joint RVs to be efficiently modelled by subsets of the data. This is the basis of **probabilistic graphical models (PGMs)**, which we explore in detail in the next section. We will study several examples of PGMs for practical ML algorithms in later sections.

15.3. Probabilistic graphical models (PGMs)

Given a probability distribution with multiple RVs, we can represent their mutual conditional independence graphically using a probabilistic graphical model (PGM). This is an **directed acyclic graph (DAG)** in which the **nodes** in the graph are the RVs such as X, Y, Z etc. and the **edges** $X \to Y$ represent the conditional dependence relationship, $P(Y|X)$, between them. We say that X is a **parent** of Y in the graph, and Y is a **child** of X. Stated another (equivalent) way, which is often more useful for modelling purposes in ML, if V has some parent variables in a PGM, but there is no edge $U \to V$, then V is **conditionally independent** of U, **given its parents**. PGMs have widespread use in probabilistic AI and ML, there are special graph structures for applications such as time series or image data. See Figure 15.2 for a simple example PGM.

Every joint distribution over a set of multiple RVs, can be expressed using the **chain rule of probability**[4], which is a product of their conditionals. For example, in the three variable case there are six different 'chain-factored' expressions,

$$\begin{aligned} P(X,Y,Z) &= P(X|Y,Z)\,P(Y|Z)\,P(Z) \\ &= P(Y|X,Z)\,P(X|Z)\,P(Z) \\ &= P(Y|Z,X)\,P(Z|X)\,P(X) \\ &= P(Z|Y,X)\,P(Y|X)\,P(X) \\ &= P(Z|X,Y)\,P(X|Y)\,P(Y) \\ &= P(X|Z,Y)\,P(Z|Y)\,P(Y)\,. \end{aligned} \tag{15.7}$$

[4]Not to be confused with the chain rule of distribution!

In fact for N variables there are $N!$ such factorizations, one for every **permutation** of the variables. Note that these factorizations of the joint expression into products of conditionals, are true regardless of the conditional independence properties of their joint distribution.[5] What makes this chain factorization useful is when it is combined with the parent/child structure of the graph and the conditional independence properties it dictates. We can find a sequence of variables e.g. $[X, U, V]$ or $[U, V, X]$, such that, every variable in the sequence occurs **after its parents** (or before its children), in the DAG. Such a sequence (there can be more than one of them) is called a **topological ordering** of the DAG. For instance, in Figure 15.2, a topological ordering is $[V, U, X, Y, Z]$. This ordering is a permutation, so we can now write down the corresponding chain factorization according to that permutation. For instance, according to the permutation $[V, U, X, Y, Z]$, we have the specific chain factorization,

$$P(V, U, X, Y, Z) = P(V)\, P(U|V)\, P(X|U, V)\, P(Y|X, U, V)\, P(Z|Y, X, U, V)\,. \tag{15.8}$$

However, note that, according to the conditional independence properties implied by the PGM, we can simplify some of the above conditionals,

$$\begin{aligned} P(Z|Y, X, U, V) &= P(Z|U, V) \\ P(Y|X, U, V) &= P(Y|X) \\ P(X|U, V) &= P(X|U) \end{aligned} \tag{15.9}$$

which can now be inserted back into (15.8), to obtain the simpler factorization,

$$P(V, U, X, Y, Z) = P(V)\, P(U|V)\, P(X|U)\, P(Y|X)\, P(Z|U, V)\,. \tag{15.10}$$

This factorization (which is unique)[6], is known as the **Markov factorization** of the PGM. Given the PGM, we can simply 'read off' this factorization directly.

Learning outcomes for section 15

- ▷ Demonstrate knowledge of the fundamental principles of probability and probability distributions: continuous and discrete sample spaces, random variables, distribution functions, normalization, joint random variables, marginalization, conditioning, chain rule.
- ▷ Explain the form and meaning of the distribution function for Bernoulli, categorical, binomial and Gaussian distributions.
- ▷ Demonstrate understanding of conditional independence and probabilistic graphical models: be able to factorize a joint distribution according to a graphical model.

[5] They simply follow from the definition of conditionals, e.g. $P(X, Y) = P(X|Y)\, P(Y)$ because $P(X|Y) = P(X, Y)/P(Y)$ so $P(X, Y) = P(X, Y) \times P(Y)/P(Y) = P(X, Y)$ as required.

[6] But note that the *ordering* of the terms in the expression is not unique, due to the commutativity of multiplication.

Section 16.

Bayesian models

Relevant reference reading material for this section is **MLSP**, Section 1.4, **PRML**, Section 1.2 and **R&N**, Section 20.2.2.

16.1. Bayes' theorem

Given a conditional distribution and the corresponding marginal for that variable, we can swap conditionals,

$$P(X|Y) = \frac{P(Y|X)\,P(X)}{P(Y)}, \tag{16.1}$$

and similar for $P(Y|X)$. This is known as **Bayes' theorem** and is so widely used that the terms in this equation have special names. $P(X)$ is called the **prior**, $P(Y|X)$ the **likelihood**, $P(Y)$ **evidence** and $P(X|Y)$ is the **posterior**. For practical reasons, if we do not know the evidence distribution, it is often presented in the following, equivalent form,[1]

$$P(X|Y) = \frac{P(Y|X)\,P(X)}{\sum_{x\in\Omega_X} P(Y|X=x)\,P(X=x)}. \tag{16.2}$$

Bayes' theorem is just a mathematical fact, but it has critical applications for rational decision-making under uncertainty in probabilistic AI and ML. Here is an example which illustrates the point. Consider the problem of determining probabilities of a range of health states of an individual, given a symptom. Precision computations are required because one of the outcomes is meningitis which is a life-threatening condition: it should be quickly treated and will often require hospitalization, but again a false positive has substantial cost.

To set up a probabilistic model, assume an RV $S = 0$ if an individual does not have a specific symptom and $S = 1$ if they have that symptom (S is Bernoulli-distributed with sample space $\Omega_S = \{0, 1\}$). This symptom could prompt three possible outcomes – $D = h$, healthy, $D = c$ influenza and $D = m$ for meningitis (D has a categorical distribution with sample space $\Omega_D = \{h, c, m\}$). Assume the likelihoods are known, i.e. the probability of a specific health state given the symptom is present:

$$\begin{aligned} P(S=1|D=h) &= 0.1 \\ P(S=1|D=c) &= 0.5 \\ P(S=1|D=m) &= 0.9. \end{aligned} \tag{16.3}$$

From the general medical literature, the priors for the health state are known (that is, the background probability of each health state in the population at large):

[1] To see why this is correct, note that $P(Y|X)\,P(X) = P(Y, X)$ so it is just the marginal.

$$\begin{aligned} P(D=h) &= 0.9 \\ P(D=c) &= 0.09 \\ P(D=m) &= 0.01. \end{aligned} \tag{16.4}$$

This reflects the fact that at any one time, most people do not have influenza, and meningitis is particularly rare. Obviously, the symptom is dependent upon the health state. This information is enough to construct a two-node PGM with:

$$P(S,D) = P(S|D)\,P(D)\,. \tag{16.5}$$

Now, using Bayes' theorem (16.1), we can compute the posterior probability of each health state, given that an individual has the symptom:

$$P(D|S=1) = \frac{P(S=1|D)\,P(D)}{P(S=1)}. \tag{16.6}$$

The evidence (the marginal symptom probability) $P(S=1)$ is not known, but (16.2) can be used instead,

$$\begin{aligned} P(S=1) = {} & P(S=1|D=h)\,P(D=h) + P(S=1|D=c)\,P(D=c) \\ & + P(S=1|D=m)\,P(D=m) \end{aligned} \tag{16.7}$$

After some arithmetic, the result is $P(S=1) = 0.14$ from which we get the posterior probabilities $P(D=h|S=1) = 0.63$, $P(D=c|S=1) = 0.31$ and $P(D=m|S=1) = 0.06$.

Initially, going on the prior information (16.4) alone would not suggest any cause for alarm. However, on observing the likelihoods, ranked in order of most probable outcome, while being healthy is still the most probable decision which the likelihoods do not change, both influenza and meningitis posterior probabilities are substantially higher. For instance, although the prior probability of meningitis is only 1%, this has been multiplied six-fold after considering the high likelihood of observing the symptom if the individual actually has meningitis. Bayes' theorem is thus a rational process which precisely synthesizes disparate sources of uncertain information.

16.2. Probabilistic classification using Bayes' theorem

Probabilistic classification can be expressed as an application of Bayes' rule: given some input (feature) data $X = x$, determine the probability $P(Y|X)$ of the class $Y = y$ for $y \in \Omega_Y$ (where Ω_Y is a discrete sample space) to which X belongs (posterior), taking into account $P(Y)$ (prior) how probable that class is before having seen the data $P(X|Y)$. A rational decision is to select the value of Y which maximizes the posterior $P(Y|X)$, which is called the **maximum a-posteriori (MAP)** decision rule:

$$y^\star = \arg\max_{y \in \Omega_Y} P(Y=y|X=x) \tag{16.8}$$

Applying Bayes' theorem to this equation we can express this as,

$$y^\star = \arg\max_{y \in \Omega_Y} P(X=x|Y=y)\,P(Y=y) \tag{16.9}$$

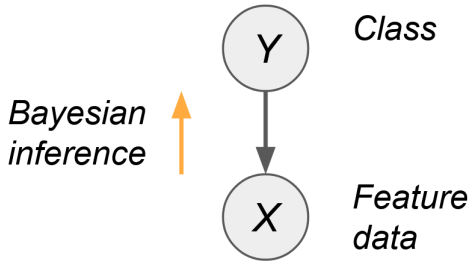

Figure 16.1.: Explaining Bayesian probabilistic inference in machine learning, for example in the context of classification. Whereas (as the PGM indicates) the computation of feature data depends upon knowledge of the class, Bayesian inference such as MAP decision-making, computes information about the class knowing the feature data, thus reversing the edge direction (orange arrow).

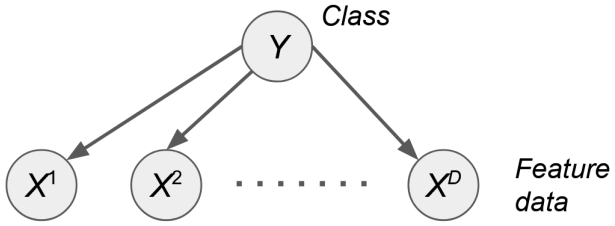

Figure 16.2.: The probabilistic graphical model (PGM) of the naive Bayes classifier, which simplifies Bayesian classification by assuming each feature dimension, $X^1, X^2, \ldots, X^D$ is independent of the others, given the class.

where the denominator in (16.2) can be omitted because the evidence probability $P(X = x)$ does not depend upon y.[2] We can view this decision as a form of **probabilistic inference** (as contrasted with logical inference in symbolic AI), where Bayes' rule is used to reverse the direction of the flow of computation (feature from class, rather than class from feature, Figure 16.1).

Using (16.9) in practical applications requires knowing the likelihood distribution $P(X|Y)$. In general, the input feature data X can be multidimensional, i.e. a D-dimensional vector, and there is no guarantee that these dimensions will be independent of one another. This makes it challenging to estimate the likelihood from data. As a convenient simplification, the **naive Bayes' classifier** assumes that each feature is conditionally independent of the others, given the class. The corresponding PGM is shown in Figure 16.2.

Reading off the Markov factorization from the naive Bayes' PGM, we have,

$$P(X|Y) = P\left(X^1|Y\right) \times P\left(X^2|Y\right) \times \cdots \times P\left(X^D|Y\right). \tag{16.10}$$

[2] This is actually a general fact about arg max and arg min: they are unchanged if the quantity being maximized is multiplied (or divided) by a value which is independent of the variable over which the quantity is being maximized.

Now, insert this quantity into Bayes' rule, to obtain,[3]

$$P(Y|X) = \frac{P(X^1|Y) \times P(X^2|Y) \times \cdots \times P\left(X^D|Y\right) \times P(Y)}{P(X)} \propto P\left(X^1|Y\right) \times P\left(X^2|Y\right) \times \cdots \times P\left(X^D|Y\right) \times P(Y) \tag{16.11}$$

and as previously, since $P(X)$ is independent of y,

$$y^\star = \underset{y \in \Omega_Y}{\arg\max} \left(P\left(X^1 = x^1|Y = y\right) \times P\left(X^2 = x^2|Y = y\right) \times \cdots \times P\left(X^D = x^D|Y = y\right) \times P(Y = y) \right). \tag{16.12}$$

It is worth interpreting the meaning of this equation. Given some new test data as a set of values from the vector $x = \left[x^1, x^2, \ldots, x^D\right]$, for each possible class $y \in \Omega_Y$, the product of the conditional likelihood $P(X|Y)$ is evaluated, and they are all multiplied together along with the class prior probability, $P(Y)$. Then the class which has the highest likelihood probability when evaluated on the test data x, *weighted by the prior for that class*, is selected as the classification decision associated with the given test data. So, if two classes have similar likelihood on the test data, the decision about which one to choose will be dominated by the relative probability of their priors. On the other hand, if the class priors are very similar, it is their relative likelihoods which most influence the class selection. This is another example of the rational fusion of different sources of uncertain information which is the hallmark of Bayesian inference in ML.

Naive Bayes' is a surprisingly good classifier for high-dimensional problems (where D is large), since it does not require a large amount of training data. Estimating feature distribution parameters is very quick, in fact, $O(D)$. Making a prediction requires evaluating D times $|\Omega_Y|$ (the number of classes), which is usually easy to carry out in practice. Nonetheless, the assumption of conditional feature independence is unrealistic for many practical ML problems.

16.3. Example: identifying pulsars

Pulsars are strange, extremely dense, astronomical objects that can be formed from **neutron stars** that spin rapidly, emitting beams of electromagnetic radiation as they do. They are called pulsars because the signal they emit sweeps across a telescopes' field of view creating a 'pulsating' (oscillating) signal. Detecting pulsars requires finding their signature in astronomical observations, but most detected signals which could be pulsars are just some kind of spurious radio interference masquerading as genuine pulsars. The problem of finding genuine pulsars from among highly diverse radio noise is not easily posed as a simple software problem because the interference can be arbitrarily complex and pulsars have multiple different sorts of oscillation 'signatures'. Furthermore, in modern digital sky surveys it is possible to identify many thousands of candidate signals which could be pulsars; sorting out spurious noise from actual pulsars is a laborious process. This is an ideal task for supervised, probabilistic ML classification because the genuine and spurious pulsar signatures have a fairly significant level of uncertainty but it is relatively easy to check pulsars by hand which can then be used as training data for a Bayesian classifier.

To apply the naive Bayes classifier to the HTRU2 pulsar dataset[4] we first need to decide on the conditional models for each putative pulsar feature. For simplicity we will use Gaussian distribution models because most of the features commonly used to identify pulsars are continuous. In the case of

[3] This can also be expressed using iterated products as $P(Y|X) = \prod_{d=1}^{D} P\left(X^d|Y\right) P(Y)$.

[4] https://archive.ics.uci.edu/dataset/372/htru2

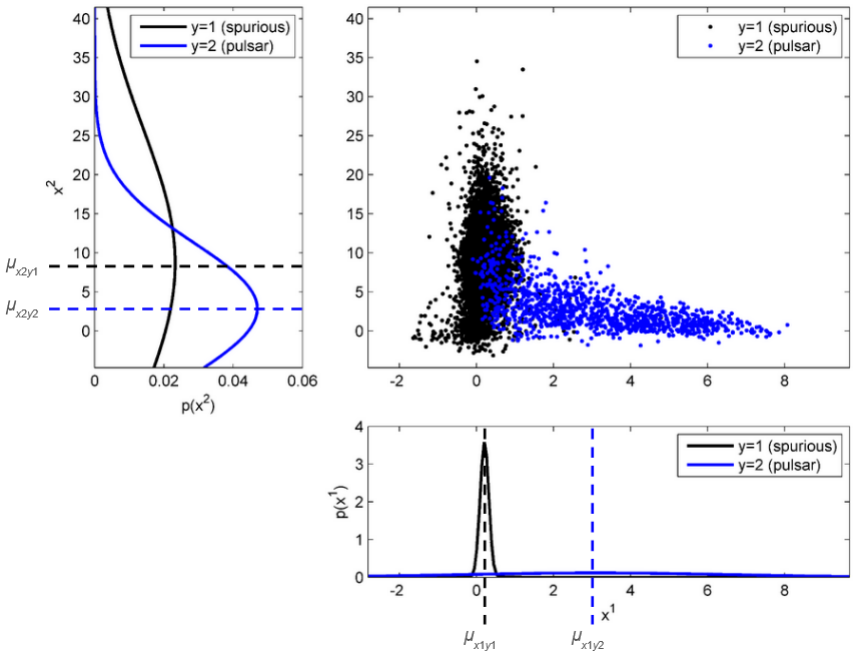

Figure 16.3.: Using naive Bayes' classification to solve the problem of automatically detecting pulsars among spurious radio interference from the automated High Time Resolution Universe digital sky survey. The conditional class likelihood models for each of the two features are Gaussian, where the means of each Gaussian under the two detection condition classes ($y = 1$, spurious interference and $y = 2$, genuine pulsar) are shown in the distribution panels. The centre panel shows a point in X^1-X^2 space for each potential pulsar candidate in the dataset.

two features, X^1 (excess kurtosis of the integrated candidate pulsar profile) and X^2 (excess kurtosis of the DM-SNR curve), we need the mean and variance parameters for each, conditional on each class, $Y = 1$ (spurious) and $Y = 2$ (pulsar). Splitting the training data into two halves, the estimate of on the training set gives, for feature X^1 in the spurious and pulsar cases, $\mu_{x1y1} = 0.2$ and $\mu_{x2y1} = 8.70$, respectively. For feature X^2, the same means are $\mu_{x1y2} = 3.1$ and $\mu_{x2y2} = 2.86$. Similarly, for variances the parameters are $\sigma^2_{x1y1} = 0.11$, $\sigma^2_{x2y1} = 17.15$, $\sigma^2_{x1y2} = 3.49$ and $\sigma^2_{x2y2} = 8.50$. The class priors are just $p_{y1} = 0.87$ and $p_{y2} = 0.13$. We can see immediately very substantial class imbalance in this case (because most of the potential pulsar signals are just radio interference). It is important to visualize the training dataset and conditionally independent feature models together (Figure 16.3).

Having these feature model parameters immediately allows computation of the test set performance of the naive Bayes' classifier, by applying the inference formula (16.12) to obtain the most probable class $y_i^\star$ for each data set item $x = [x^1, x^2]$ in the test split of the dataset. The 0-1 classification error can then be calculated as $F_{01} = \sum_{i=1}^{N} \mathbb{I}\left[y_i^\star \neq y_i\right]$ where y_i is the known label. In this case, the naive Bayes' classifier has $F_{01} = 1181$ classification errors in test set size $N = 8950$ candidates, giving a test set accuracy of approximately 80%.

This is quite good accuracy for such a simple model, but not sufficiently accurate for scientific purposes in astronomy. It would be reasonable to trust those pulsar candidates with high posterior probability, which can be calculated by dividing through by the evidence probability $P\left(x^1, x^2\right) = \sum_{y\in\{0,1\}} P\left(X^1 = x^1 | Y = y\right) P\left(Y = y\right)$ in each test case. In fact for this model, 91% of the predicted cases have posterior probability of 75% or higher, thus leaving only about 10% of cases which might

need to be checked by hand (i.e. about 900 cases out of nearly 9000), very substantially reducing the labour required to obtain good quality scientific evidence.

Learning outcomes for section 16

- ▷ Show understanding of Bayes' theorem: prior, posterior, evidence and likelihood distributions, swapping conditionals.
- ▷ Demonstrate the principle of MAP inference as applied to probabilistic classification.
- ▷ Explain the rational behind, probabilistic structure of, and method of, naive Bayes classification: apply naive Bayes' to small classification problems.

Section 17.

Sequential problems and hidden Markov models

Relevant to this section is the material in **PRML**, Section 13.2, **MLSP**, Section 9.4 and **R&N**, Section 15.3 (matrix algebra approach to HMMs).

17.1. Markov chains and the hidden Markov model

Many applications of ML involve **ordered data** that is, data for which the **ordering** matters. Examples include:

- ▷ **Natural language** (ordered sequences of words)
- ▷ **Appointment calendar entries** (date and time-ordered event names)
- ▷ **Medical symptom diaries** (e.g. time-ordered records of pain measurements)
- ▷ **Electronic health records** (time-ordered sequences of medical system interactions)
- ▷ **Macroeconomic time series** (time-ordered sequences of GDP values)
- ▷ **Genomics** (base pairs in a genome sequence)

What has been shown is that ML algorithms such as regression and classification are more useful if they take into account this ordering. For probabilistic AI/ML this requires special kinds of PGMs, a primary example of which is the so-called **Markov chain** in which a sequence of RVs $Y_0, Y_1, \ldots, Y_T$ is organized such that the next one in the sequence only depends upon the previous one, i.e. the PGM where,

$$P\left(Y_t|Y_{t-1}, Y_{t-2}, \ldots, Y_0\right) = P\left(Y_t|Y_{t-1}\right), \tag{17.1}$$

for all $t = 1, 2, \ldots, T$, along with the independent Y_0 to get the iteration started. If there are many sequences of data then (17.1) can be fitted to the data and used to make predictions. In practical applications, however, it is more common that the sequence $Y_0, Y_1, \ldots, Y_T$ is not actually measured. The appropriate PGM for this situation is called the **hidden Markov model (HMM)** (Figure 17.1), where there is an edge $Y_{t-1} \to Y_t$ and another $Y_t \to X_t$ are called **hidden states** and the variables Y_t the **observations**. Typically, the hidden states are discrete with finite sample space Ω_Y. The observations therefore only depend upon the hidden state at the same time instant.

In applications where HMM are used, there is the need to solve the following problems:

- ▷ **Model fitting**: given only the sequence of observed data $X_0, X_1, \ldots, X_T$, estimate the distribution functions in the PGM, i.e. find $P\left(X_t|Y_t\right)$ (the distribution which models how the observed data depends upon the hidden states) and $P\left(Y_t|Y_{t-1}\right)$ (the distribution of the current state given the previous one),

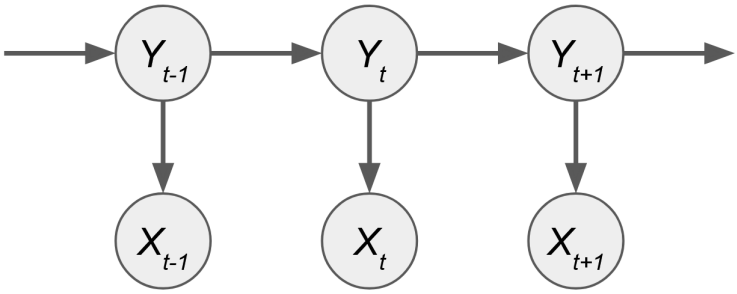

Figure 17.1.: The probabilistic graphical model (PGM) of the hidden Markov model (HMM).

▷ **Evaluation**: given fixed model parameters and complete sequence of observed data, compute the probability of the complete sequence of observed data, $P(X)$, and,

▷ **Decoding**: given fixed model parameters, and data compute the most probable sequence of hidden states, $y^\star = [y_0^\star, y_1^\star, \ldots, y_T^\star]$.

On the face of it, solving these problems requires evaluating *all possible sequences of hidden states* which we denote by $\mathcal{Y}$; if there are K hidden states, this means $O\left(K^T\right)$ (exponential complexity). However, the use of dynamic programming (Section 5) makes this tractable, leading to efficient, linear-time $O\left(TK^2\right)$ inference algorithms.

17.2. Viterbi decoding

This course will investigate the efficient **Viterbi decoding** algorithm for HMMs. Stated mathematically, the problem to be solved is,

$$P(y^\star) = \max_{y' \in \mathcal{Y}} P\left(X_0 = x_0, \ldots, X_T = x_T, Y_0 = y'_0, \ldots, Y_T = y'_T\right), \tag{17.2}$$

where $x_0, x_1, \ldots, x_T$ is the set of observed data, $y = [y_0, y_1, \ldots, y_T]$ is a hidden sequence, and the configuration space $\mathcal{Y}$ is the set of all possible such sequences of hidden states over time $t = 0$ to $t = T$. For example, with $\Omega_Y = \{1, 2, 3\}$ so that $K = 3$ and $T = 3$, then particular hidden sequences are $y = [1, 2, 1, 3]$ and $y = [3, 2, 2, 1]$. The hidden state can change from any of the K states at time t, to any of the K states at time $t + 1$. So, if there are $t = 0, 1, \ldots, T$ time stages, there are K^{T+1} possible sequences, an intractable exponential number of possible sequences over which to maximize (17.2).

However, there is a factorized SDP for these hidden sequences, through which we can use dynamic programming to make this maximization tractable. Assume we have the sequence of states for stage $t-1$. The K new sequences of states at stage t, is simply obtained from each sequence at stage $t-1$, by appending each of the K possible states. Now, considering the HMM graphical model up to stage $t-1$, associated with the sequence of states is the (joint) optimal distribution (for clarity, we drop the assignment of observed data to their random variables),

$$P^\star\left(X_0, \ldots, X_{t-1}, Y_{t-1}\right) = \max_{y' \in \mathcal{Y}_{t-2}} P\left(X_0, \ldots, X_{t-1}, Y_0 = y'_0, \ldots, Y_{t-2} = y'_{t-2}, Y_{t-1}\right), \tag{17.3}$$

where $\mathcal{Y}_{t-2}$ is the set of all possible hidden sequences up to time $t-2$.

According to the SDP above, associated to every one of the newly generated hidden sequences at each value of $y \in \Omega_Y$, is the probability obtained by appending the variables $X_t = x_t$ and $Y_t = y$, to the PGM at stage $t-1$, so that,

$$P\left(X_0, \ldots, X_{t-1}, X_t, Y_{t-1}, Y_t = y\right) = P\left(X_0, \ldots, X_{t-1}, Y_{t-1}\right) P\left(Y_t = y | Y_{t-1}\right) P\left(X_t | Y_t = y\right). \quad (17.4)$$

Maximizing over Y_t leads to,

$$P^\star\left(X_0, \ldots, X_t, Y_t = y\right) = \max_{y' \in \Omega_Y} P^\star\left(X_0, X_1, \ldots, X_{t-1}, Y_{t-1} = y'\right) P\left(Y_t = y | Y_{t-1} = y'\right) P\left(X_t | Y_t = y\right), \quad (17.5)$$

and writing $p_t^\star(y) = P^\star\left(X_0, \ldots, X_t, Y_t = y\right)$ shows that the above is in the form of a Bellman recursion,[1]

$$p_t^\star(y) = \max_{y' \in \Omega_Y} p_{t-1}^\star(y') P\left(Y_t = y | Y_{t-1} = y'\right) P\left(X_t | Y_t = y\right). \quad (17.6)$$

At the final stage $t = T$,

$$\begin{aligned} \max_{y \in \Omega_Y} p_T^\star(y) &= \max_{y' \in \mathcal{Y}_T} P\left(X_0 = x_0, \ldots, X_T = x_T, Y_0 = y_0', \ldots, Y_{T-1} = y_{T-1}', Y_T = y\right) \\ &= P\left(y^\star\right), \end{aligned} \quad (17.7)$$

is the required solution to the HMM optimization problem (17.2). This shows how the optimization problem can be solved efficiently by maximizing over each successive time step, which is an $O\left(TK^2\right)$ process, i.e. linear time computation. This process of generating new state sequences by appending states, and corresponding joining of PGMs together which makes the DP recursion possible, is illustrated in Figure 17.2.

To get the recursion (17.6) started, we only need the initial probability function, $p_0^\star(y) = P\left(X_0 | Y_0 = y\right) P\left(Y_0 = y\right)$. The optimal sequence of states $y^\star$ can be reconstructed by (a) retaining the optimal decision at each stage, (b) solving for $y_T^\star$ by optimizing over $p_T^\star(y)$, then (c) using the retained decisions to work backwards from $y_T^\star$. Putting this together leads to Viterbi decoding, Algorithm 17.1.

17.3. Example: genome sequence region segmentation

Genome sequences are made from four DNA base pair molecules: adenosine (A), cytosine (C), guanine (G) and thymine (T). Sub-sequences of three ACGT base pairs encode for different amino acids, which ultimately go to make up proteins which are the building blocks of every part of the living cell. Different sub-sequences of the DNA sequence have different functions in this process, which involves splicing together coding regions called exons (E), and discarding introns (I) and untranslated 5' donor site (D) sub-sequences.

For molecular biology it is very useful to be able to segment a sequence into these diferent sub-sequences. As the relationship between DNA sequences and coding/non-coding regions is complex, it is well-modelled as an HMM, where the hidden states $\Omega_Y = \{e, d, i\}$ are the $K = 3$ possible regions,

[1] This function has various names in the literature, including the *forward optimal message* $\alpha(y)$.

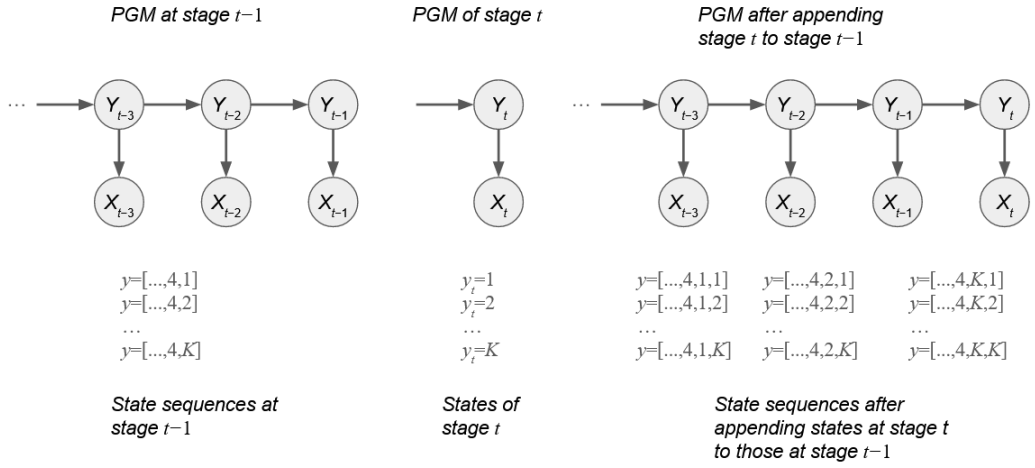

Figure 17.2.: Illustrating the relationship between the SDP which generates new hidden state sequences from previous ones through appending states, and the corresponding process of appending variables and their Markov dependencies, to the PGM to create new, extended PGMs and associated distribution function.

Algorithm 17.1 The Viterbi decoding algorithm for determining the optimal sequence of hidden states in an HMM.

▷ **Step 1**. *Initialization*: Compute the initial optimal probability function, $p_0^\star(y) = P(X_0 = x_0|Y_0 = y)\, P(Y_0 = y)$,

▷ **Step 2**. *Forward recursion*: Compute the sequence of optimal probability function while keeping track of the corresponding argument function,

$$p_t^\star(y) = \max_{y' \in \Omega_Y} p_{t-1}^\star(y')\, P(Y_t = y|Y_{t-1} = y')\, P(X_t = x_t|Y_t = y)$$
$$Y_t^\star(y) = \arg\max_{y' \in \Omega_Y} p_{t-1}^\star(y')\, P(Y_t = y|Y_{t-1} = y'),$$

for $t = 1, 2, \ldots, T$,

▷ **Step 3**. *Backtracking*: Reconstruct the best sequence of hidden states in reverse,

$$y_T^\star = \arg\max_{y \in \Omega_Y} p_T^\star(y),\ y_{t-1}^\star = Y_t^\star(y_t^\star),$$

for $t = T-1, T-2, \ldots, 0$.

Stage	x_t	$p_t^\star(y=e)$	$p_t^\star(y=d)$	$p_t^\star(y=i)$	$Y_t^\star(y=e)$	$Y_t^\star(y=d)$	$Y_t^\star(y=i)$	$y_t^\star$
$t=0$	g	0.2500	0.0000	0.0000				e
$t=1$	g	0.0563	0.0238	0.0000	e	e	e	e
$t=2$	g	0.0127	0.0053	0.0024	e	e	d	e
$t=3$	g	0.0028	0.0012	0.0005	e	e	d	d
$t=4$	t	0.0006	0.0000	0.0005	e	e	d	i
$t=5$	a	0.0001	0.0000	0.0002	e	e	i	i

Table 17.1.: Example run of Viterbi decoding, Algorithm 17.1, applied to the problem of segmenting DNA sequences of ACGT base pairs, into introns (i), exons (e) and 5' donor site (d) sub-sequences. The HMM hidden state transition $P(Y=y|Y=y')$, observation distributions $P(X=x|Y=y)$ and initial distribution $P(Y_0=y)$ are given in the text.

and the DNA sequence is the observed data $\Omega_X=\{a,c,g,t\}$. Knowledge of the molecular biology and experimental work has determined the following state transition and observation distributions,

$$P(Y=y|Y=y') = \begin{bmatrix} & y=e & y=d & y=i \\ y'=e & 0.9 & 0.1 & 0 \\ y'=d & 0 & 0 & 1.0 \\ y'=i & 0.1 & 0 & 0.9 \end{bmatrix}$$
$$P(X=x|Y=y) = \begin{bmatrix} & x=a & x=c & x=g & x=t \\ y=e & 0.25 & 0.25 & 0.25 & 0.25 \\ y=d & 0.05 & 0 & 0.95 & 0 \\ y=i & 0.4 & 0.1 & 0.1 & 0.4 \end{bmatrix} \tag{17.8}$$

and the assumption is that the sequence always starts at an exon, e.g. $P(Y_0=e)=1$, $P(Y_0=d)=0$ and $P(Y_0=i)=0$. This encodes the known biological facts that exons can be followed by donor sites, but not directly by introns; and donor sites can only be followed (immediately) by introns etc. Donor sites consist only of AG base pairs. This is enough information to run the Viterbi algorithm on a given DNA sequence to determine the optimal sequence of segments (Table 17.1). Since this DP algorithm runs in linear time, it can be easily applied to the usually extremely long DNA sequences encountered in biology (i.e. the human genome has about 6 billion base pairs), and is a good example of the ubiquitous use of ML in modern bioinformatics.

An important computational point needs to be raised. The optimal probabilities $p_t^\star(y)$ eventually become extremely small and difficult to work with computationally, this is a consequence of long sequences leading to long iterated products of probabilities in the recursion (17.6). A practical solution is to work with **log-probabilities** instead, that is, first take the logarithm of every probability, then since $\ln(p\times q)=\ln p+\ln q$, the recursion involves sums of log-probabilities rather than products, and the Viterbi decoding is otherwise unchanged.[2]

[2]This works because the maximum distributes over sum as it does over the product, the condition required for the principle of optimality to hold.

Learning outcomes for section 17

- ▷ Demonstrate knowledge of the structure of the HMM: Markov chains and hidden states, relationship to observed data.
- ▷ Explain the function of the steps of the Viterbi decoding algorithm.
- ▷ Apply the Viterbi decoding algorithm to small sequence decoding problems.

Section 18.

Other sequential models

Relevant reference reading material for this section can be found in **MLSP**, Section 7.5, **PRML**, Section 13.3 and **R&N**, Section 15, Section 17.

18.1. Optimal sequential decision-making

Making a sequence of choices based on observed effects of those choices, is a common problem in real-world planning and decision-making. Ideally, we want to decide on a sequence of actions which are **optimal** (the best possible decisions which can be made) with respect to the available information about the effects of these decisions. The problem of computing the optimal decision sequence or **policy**, can make use of dynamic programming applied to a special kind of sequential graphical model, known as a **Markov decision process (MDP)** (Figure 18.1).

An MDP represents a sequence of input **actions** A_t which lead to a sequence of observed **states**, S_t for $t = 1, 2, \ldots, T$. Each state has an associated **reward** $R(S_t)$ which is a deterministic function of the state.[1] A sequence of states has a **utility** which is a deterministic function of all these states $U(S_1, \ldots, S_T)$. Methods such as **reinforcement learning (RL)** uses observed rewards which are a function of the MDP state, to learn an optimal action sequence (policy), and is mainstay of modern AI in, for instance robotics and fine-tuning of large language models.

[1] Deterministic means not random (although in this case, since the state itself is random, the reward as a function of the state is random).

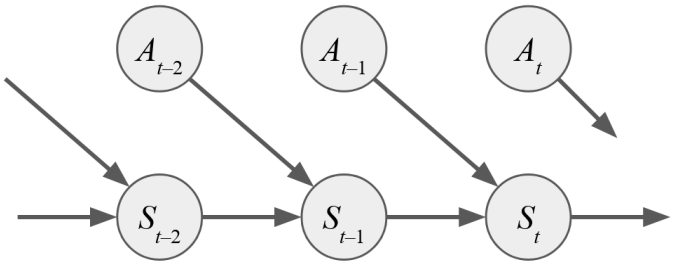

Figure 18.1.: Markov decision processes (MDP) are probabilsitic graphical models in which a sequence of actions A_t influences a sequence of observed states S_t in the world, which themselves also depend upon previous states.

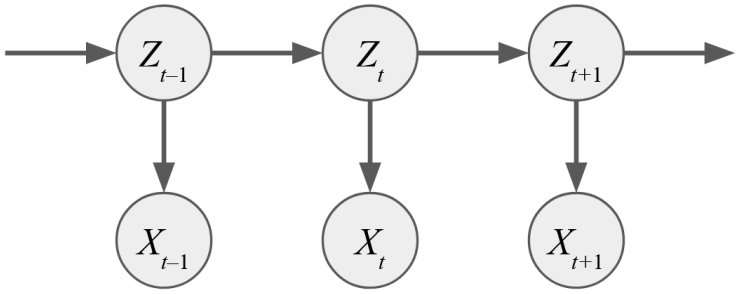

Figure 18.2.: The Kalman filter is a probabilistic model which captures dependence in time for a continuous state variable Z_t, upon which noisy observations of the state X_t depend. This has identical probabilistic dependence structure to the HMM.

18.2. Kalman filter

Similar to HMMs but for situations with continuous hidden state, the **Kalman filter** is often used for **target tracking** or **motion smoothing** with noisy observations. This has a wide array of applications in, for instance, automated airplane (autopilot) or ship guidance. For the **linear-Gaussian** case[2], the Kalman filter can be conceptualized as a **predictor-corrector process** to estimate the hidden state, which alternates between the two steps:

1. Using a model, predict the next state from the current estimate of the state,
2. When the next noisy observation of the state becomes available, correct the last prediction using a suitably weighted average of the noisy observation and the previous prediction.

It turns out that the weighted average arises directly from the use of Bayes' theorem, which optimally balances the uncertainty in the predictions, against the uncertainty in the noisy observations.

The Kalman filter captures dependence in time which is not directly observed, each continuous hidden state Z_t depends only upon the one before it in time, Z_{t-1} for all $t = 1, 2, \ldots, T$. The noisy observations X_t depend only upon the associated hidden state, Z_t (Figure 18.2). The simplest (and most widely encountered) model is a linear-Gaussian recurrence relation:

$$\begin{aligned} Z_t &= AZ_{t-1} + U \\ X_t &= BZ_t + V, \end{aligned} \tag{18.1}$$

for all $t = 1, 2, \ldots, T$ and where matrices A, B are **state update** and **observation models**, and U, V are Gaussian error/noise vectors.

18.3. Recurrent neural networks (RNN)

'Standard' deep nets discussed in section 13, cannot explicitly take data ordering into account. **Recurrent neural networks (RNN)** treat the hidden layer in the previous stage as an additional input to current stage in the ordering. The output of the previous stage's hidden layer z_{t-1} is included with computation of the current hidden layer, z_t. In this way, order dependence is explicitly taken into account in the network structure (Figure 18.3). This defines a (nonlinear) recurrence relation,

[2]Models which are linear functions of the random variables, which are all (multivariate) Gaussian.

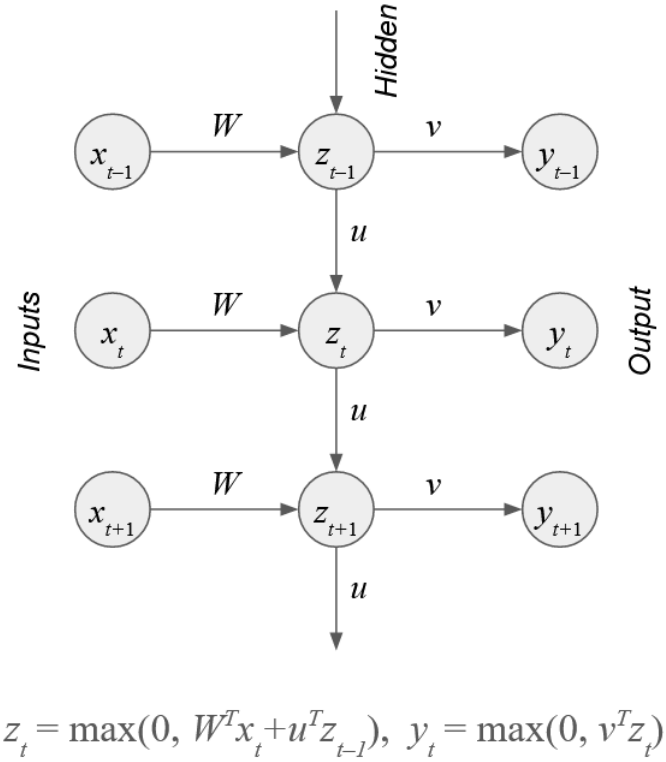

Figure 18.3.: Recurrent neural networks (RNNs) are deep networks which take into account time ordering of input data x_t by treating the output of the hidden layer in stage $t-1$ as an additional input to the the next stage's network.

$$\begin{aligned} z_t &= \max\left(0, W^T x_t + u^T z_{t-1}\right) \\ y_t &= \max\left(0, v^T z_t\right), \end{aligned} \tag{18.2}$$

for $t = 1, 2, \ldots, T$ with the additional state z_0 required to get the recursion started.

RNNs are widely used in modern AI applications such as **seq2seq** networks which map a source sequence into a target sequence, taking into account the sequential dependency in the inputs and outputs. Examples of applications where such networks are used include **neural machine translation** of text in one language to another language, and forms the basis of some **chatbots** and **large language models**.

Learning outcomes for section 18

- ▷ Define Markov decision processes, Kalman filters and recurrent neural networks: explain their significance for applications.
- ▷ Recognize and provide the mathematical structure of these sequential models.

Terminology

Activation function	In deep learning, each neuron has a nonlinear activation function acting on the accumulated inputs to create its output.
AI (artificial intelligence)	Computational algorithms for solving complex, real-world problems.
Algorithm strategy	A systematic principle for solving a broad class of AI/ML problems.
Annealing schedule	In the simulated annealing method for combinatorial optimization, the annealing schedule parameter determines for how long bad steps which worsen the objective, will be taken.
Approximate method/algorithm	A computational algorithm which is only guaranteed to find a locally optimal value; this could also be globally optimal, but we cannot be sure.
Approximate optimization algorithm	An approximate algorithm finds a 'good enough' solution to an optimization problem, not one which is guaranteed to be globally optimal. It may be locally optimal.
Atomic statements	A primitive logical statement which has Boolean truth value, including the constants True and False.
Automatic differentiation (AD)	Gradients calculated exactly at a specific numerical value of the function, by breaking the function down into elementary expressions which have simple chain rule forms.
Backpropagation	Gradients of objective functions of deep neural networks can be computed by backpropagating gradients through layers using the chain rule of calculus.
Bayes' theorem	Given a conditional distribution and marginals, Bayes' theorem swaps the conditional.
Bellman recursion	A recursive mathematical equation calculating the optimal configuration at each stage of a dynamic programming algorithm, from optimal configurations in the previous stage.
Bellman's principle of optimality	Optimal solutions to optimization problems are composed of optimal solutions to sub-problems.
Big-oh notation	A function of a single parameter representing the asymptotic behaviour of all members of its complexity class.

Boolean function	Function which take Boolean inputs, producing a Boolean output. The special logical operations “and”, “or”, “not”, “implies” and “if and only if” are called connectives.
Boolean set	The set of values with two elements False, True which are used in logical computations.
Chain rule of calculus	The gradient of a function of a function is obtained using this mathematical rule.
Chain rule of probability	The distribution over every set of joint random variables can be expressed as an appropriate product of their conditionals.
Child	In a directed graph, if there is an edge $X \{}to Y$, then Y is a child of X.
Classifier (ML)	An algorithm, optimized on training data, which takes feature data as input and produces a classification decision as output.
Cluster centroid	A cluster centroid is a representative value in the space of the data around which a cluster is defined.
Clustering	A clustering of a data set is a partition of the data items into classes (clusters) such that some objective function which measures the quality of the clustering, is optimized.
Combinatorial explosion	The size of the space of all possible configurations, grows rapidly with the size of the problem.
Combinatorial neighbourhood	A set of configurations which are “close” to a given configuration, under some measure of combinatorial distance between configurations.
Complex (logical) statements	A logical statement constructed from atomic logical statements using logical connectives.
Complexity class	A measure of how rapidly the value of a function grows with the value of its input.
Computational complexity	The complexity class of the worst case number of computational steps an algorithm requires to complete.
Computational graph	A graphical representation of the computational operation of an algorithm.
Conditional distribution	Distribution arising from fixing the values of a subset of a collection of joint random variables.
Conditional independence	In a PGM, if the parents of a variable V are fixed and U is not a parent of V, then V is said to be conditionally independent of U.
Configuration	A specific state of an AI or ML problem.

Constant	A term in an algebraic expression with a fixed (semantic) value.
Continuous event	In a probabilistic model, a continuous event is a specific outcome in an uncountably infinite set, for example, the real line.
Convergence guarantee	An optimization algorithm which, when allowed to search for an infinite number of steps, is guaranteed to find the global optima, is said to have guaranteed convergence.
Convex optimization algorithm	A convex function has one minima which is the global one.
Convolutional neural network (CNN)	A deep neural network with weight sharing over 'sliding windows' of the input connections to each layer.
Decision (classification) boundary	In classification in ML, a set of (one or more) boundaries which partition the data into two or more classes. Linear classifiers have (in general) hyperplane boundaries.
Directed acyclic graph (DAG)	Nodes with directed edges between them in which there are no directed cycles, that is, no directed paths which lead from a node back to itself.
Discrete outcome	In a probabilistic model, a discrete event is a specific item in a (countable) set.
Divergent iteration	In solving an optimization problem, if the algorithm does not converge on a solution, it can instead diverge so that the algorithm never terminates.
Dual numbers	An algebraic tool for efficiently computing the gradient of composite functions using the chain rule of calculus.
Dynamic programming (DP)	An efficient algorithm strategy for solving combinatorial AI problems in polynomial time, using factorization and the principle of optimality.
Edge	A directed graph consists of points (nodes) which are joined by lines with arrowheads (edges).
Error function (ML)	In machine learning, an error function measures the extent to which an algorithm can predict the data.
Evidence distribution	In Bayes' theorem, the evidence distribution is the marginal of the likelihood variable.
Exact method/algorithm	A computational algorithm which is guaranteed to find the globally optimal value of an objective function.
Exhaustive method	Solving a problem by testing all possible configurations.
Factorization	A factorized expression uses distributivity to create bracketed expressions, for example $a \times b + a \times c$ can be factorized as $a \times (b + c)$.

Function composition	Feeding the output of one function into the input of another, and so on.
Globally optimal value	No other value of the objective function is better than the globally optimal value.
Greedy approximate search	Also known as hill-climbing or iterative improvement, an algorithm which uses successive neighbourhood search to find a local optima.
Hidden Markov model (HMM)	Markov chain where the sequence of time-ordered random variables is not directly measured and must be inferred from the observations.
Independent random variables	If the joint distribution of random variables can be factored into a product of their marginal distributions, we say that the variables are independent.
Indicator function	A function which returns the value 1 if a given logical condition is true, and 0 otherwise.
Joint random variables	Two or more random events co-occurring. Their combined distribution is called the joint distribution.
K-means algorithm	An iterative method for approximately solving the K-means clustering problem.
K-means error (objective)	A least-sum-of-squares objective measuring the quality of a clustering partition; also the name for an iterative algorithm which minimizes this objective.
Kalman filter	A linear-Gaussian sequential model with continuous hidden state and noisy observations, which optimally balances prediction error against observation error.
Learning rate (ML)	The size of each step in a gradient descent algorithm for optimization of machine learning model parameters.
Likelihood distribution	In Bayes' theorem, the conditional distribution to be reversed to obtain the posterior distribution.
Linear classifier (ML)	Classifier with hyperplane decision boundary.
Linear regression (ML)	Curve-fitting a linear function (hyperplane) of continuous parameters.
Linearly separable data	Data which can be perfectly separated into two classes by a linear classifier which has a hyperplane decision boundary.
Locally optimal value	In some part of (some subset) of the configuration space, no other value of the objective function is better than the locally optimal value in that subset.

Logic	Combining facts using unambiguous logical rules so as to infer or deduce other facts.
Logical entailment	In propositional logic, one statement P entails another Q if and only if the models satisfying Q contain all the models satisfying P.
Logical inference	Applying the rules of logic to facts, to correctly compute other facts.
Logical knowledge base	A collection of logical rules and facts, which are represented as propositional variables and expressions.
Logical model	Specific assignment of a collection of logical variables, to specific Boolean values.
Logical model checking	Testing entailment of logical propositions through exhaustive computation of their respective models.
Logical satisfication	Any proposition which is true under the conditions of some model, is said to be satisfied by (or under) that model.
Look-up table	Table in a database.
Machine learning (ML)	Algorithms whose parameters are determined through optimizing an objective function with respect to those parameters, using data from the real world.
Marginal distribution	Distribution of a subset of a collection of joint random variables.
Markov chain	A probabilistic model of joint random variables which are ordered such that the next random variable in the sequence depends only upon the previous one.
Markov decision process (MDP)	Time-dependent probabilistic model of sequential actions and their observed effects, where effects also depend upon each other sequentially.
Markov factorization	Every PGM has an equivalent representation in terms of a factored joint distribution expression for the random variables represented by the nodes in the DAG.
Maximum a-posteriori (MAP)	The value of a random variable which maximizes its posterior probability.
Mean	Average value of a distribution.
Memoization	Storing a previously computed solution to a problem in memory and then retrieving it from memory when the same problem is required subsequently.
Minimization	Finding the smallest value of.

Misclassification error	Sum of the number of data points in a data set which are incorrectly partitioned into the wrong labels by a classification algorithm, given the known labels in the data set.
ML (machine learning)	Computational algorithms for solving complex, real-world problems by learning patterns from, and then applyng them to, real-world data.
Multilayer perceptron	Perceptrons chained together so that the output of one feeds into the input of the perceptrons in the next layer.
Naive Bayes' classifier	Bayesian probabilistic classification algorithm which assumes the multidimensional feature data is independent between dimensions.
Nodes	A directed graph consists of points (nodes) which are joined by lines with arrowheads (edges).
Objective function	A mathematical function or model, which represents an AI or ML problem in the real-world.
Optimization problem	A problem which is represented as minimizing a mathematical function over its arguments.
Parent	In a directed graph, if there is an edge $X \{}to Y$, then X is a parent of Y.
Partition	Splitting a collection of objects into smaller collections, which together make up the whole collection (exhaustive) and where there is no overlap between the smaller collections (exclusive).
Perceptron	Single layer linear classifier with error function which penalizes incorrect classification decisions.
Perceptron algorithm	Sequential gradient descent for the single layer linear classifier with perceptron loss.
Perceptron error function	The objective function of the perceptron classifier.
posterior	In Bayes' theorem, the posterior distribution is the likelihood distribution with conditioning and conditioned variables swapped.
Prediction (ML)	After learning parameter values from data (training), machine learning algorithms produce an output value, known as a prediction, given new input data.
Prior distribution	In Bayes' theorem, the marginal of the conditioning variable in the likelihood.
Probabilistic graphical model (PGM)	Conditional independence relationships between joint random variables expressed using a directed acyclic graph (DAG).
Probability	Mathematically precise quantification of uncertainty.

Probability density function (PDF)	The PDF gives a probability density (that is, probability per unit of the variable) of every event for a continuous random variable.
Probability density function (PDF)	The PDF gives a probability density (that is, probability per unit of the variable) of every event for a continuous random variable.
Probability density function (PDF)	The PDF gives the amount of density per unit for every possible value of a continuous random variable.
Probability distribution function	Function which provides a probability value, either an actual probability in the discrete case, or a probability density value in the continuous case, to every outcome in a probability model.
Probability mass function (PMF)	The PMF gives an actual probability of every event for a discrete random variable.
Problem size	The number of discrete items of data or information, in a problem.
Propositional calculus	A formal algebraic system for organizing facts and performing logical computations with them so as to draw logical inferences.
Random events	Outcomes or occurrences in the real world that are treated within a mathematical model as having no known (or observable) cause.
Random variable (RV)	Function which assigns a unique numerical value to an event or outcome in a probabilistic model. These numerical values can then be used in ordinary arithmetic calculations.
Recurrent neural network (RNN)	Time-dependent deep neural network which takes the output in the hidden layer in the previous stage, as an additional input to the current stage.
Recursion	A mathematical/computational process which is repeatedly applied to its own output.
Regression (ML)	Curve fitting of a continuous-valued function of continuous parameters.
Saddle point	A point in a curve where the gradient goes through zero, but which is not a maxima or minima.
Sample space	The set of all possible events in a probability model, usually denoted $\{}Omega$.
Semantics	Rules by which variables in syntactically correct expressions are evaluated, so that the whole expression can be evaluated.
Sequential decision process (SDP)	An SDP processes the input data in some sequential order, accumulating results.

Sequential gradient descent (SGD)	SGD is a continuous optimization method which finds local optima in an error function.
Simulated annealing	Iterative improvement/greedy approximate search for combinatorial optimization, which allows occasional backwards steps to attempt to escape local optima. The probability of taking backwards steps decreases with the number of iterations performed.
Sound and complete logical inference	An algorithm which is guaranteed to draw correct inferences in every case.
Space complexity	The complexity class of the worst case amount of memory an algorithm requires.
Sum-of-squares loss	Non-negative measure of prediction error based on the square of the difference between the prediction model output and the labelled data.
Supervised learning (ML)	An algorithm which uses training feature data and associated labels, to optimize the error function with respect to the parameters.
Support vector machine (SVM)	Single layer classifiers which cope with linearly non-separable data by incorporating a margin of error around the decision boundary.
Surrogate error function	Loss function which is chosen for mathematical convenience of optimization.
Symbolic AI	AI using symbols such as text or other discrete information.
Syntax	Rules by which sequences of symbols can be organized into expressions. Syntactically correct sequences are said to be well-formed, and are valid expressions in the syntax.
Test data (ML)	Data used to estimate the predictive accuracy of a trained ML model.
Time series	Measurements of some quantity at discrete intervals in increasing time.
Topological ordering of a DAG	Sequence of nodes in a DAG such that every node occurs after its parents in the DAG, or equivalently, before its children in the DAG.
Training data (ML)	Data used to estimate the best values of the model parameters.
Truth table	For a Boolean function, a systematic list of all possible truth values of inputs, along with the truth value of the corresponding output.
Truth value	A Boolean False or True value.

Unsupervised learning (ML)	Without labels associated to a set of feature data, an unsupervised ML algorithm learns to extract patterns from the given data.
Variance	One measure of the spread of a distribution of a random variable, around the mean.
Viterbi decoding	Given a fixed HMM model and data, a dynamic programming algorithm for finding the most probable sequence of fixed states.
Weight sharing	In a deep neural network, multiple inputs to a set of neurons can have the same weight, which reduces the overall complexity of the network for special applications.