

THE LOST ALGORITHMS

Volume One — Optimisers and Predictors

By

MARTIN CARR

INDEPENDENT PUBLISHER

Copyright © 2026 by Martin Carr.

All rights reserved.

No part of this book may be reproduced, digitalized, or transmitted in any form without written permission from the publisher, except in the case of brief quotations embodied in critical reviews.

Publisher: Independent Publisher

ISBN Placeholder

TABLE OF CONTENTS

Preface: The Vault That Unlocked	4
How to Read This Book	9
A Note on Patents and the Public Domain	11
Part I — Evolution: Algorithms That Breed Solutions	14
The Machine That Bred Answers (Genetic Algorithms)	15
Programs That Write Programs (Genetic Programming)	25
The Swarm (Particle Swarm & Differential Evolution)	35
Annealing the Answer (Simulated Annealing & Ant Colony Optimisation)	42
Evolving the Network Itself (Neuroevolution)	51
Guessing Smarter (Bayesian Optimisation & Estimation of Distribution)	59
Part II — The Neuron: How Networks Learned to Learn	68
Backprop in a Box (The Modular Neural Network)	69
The GPU Prophecy (Convolutional Neural Networks on Graphics Hardware)	77
Memory in the Wires (Recurrent Networks)	85
The Descent Toolkit (SGD, Momentum, RMSProp, and Adam)	93
Part III — The Verdict: Classifiers and the Art of Deciding	102
The Margin (Support Vector Machines)	103
The Tree and the Forest (Decision Trees & Random Forests)	112
Wisdom of Weak Crowds (AdaBoost)	121
The Vote (Voting and Stacking Ensembles)	129

PREFACE: THE VAULT THAT UNLOCKED

Somewhere in a building you have never visited, in a filing cabinet that no longer exists, a clerk once stamped a date onto a document describing how to make a machine that breeds its own answers. The date mattered more than anyone in the room understood. A United States patent grants its owner a monopoly that lasts, in round terms, twenty years from the day it was filed. For twenty years the owner can stop anyone else from making, using, or selling the invention. And then — on a specific morning, without ceremony, without a press release — the monopoly ends. The invention falls into the public domain. It belongs to everyone. It belongs to you.

Between roughly 1995 and 2006, the institutions that built modern machine learning were filing patents at a furious pace. Microsoft Research, IBM, Bell Labs, AT&T, Lucent, Lockheed Martin, General Electric, a scattering of universities and a few lone inventors — they

were patenting the algorithms that now run quietly inside almost everything you touch. The recommendation engine that suggested this book. The spam filter that caught the email you never saw. The clustering that sorts your photos by face, the filter that keeps a drone steady in wind, the boosting algorithm that decides whether a transaction is fraud. The foundational versions of all of it were, at the time, somebody's property, locked behind a number and a claim.

Those twenty years are up. Every patent described in this book has expired. The ideas inside them are now public domain — free, legally and permanently, for anyone to read, build, ship, and sell. This is not a loophole or a grey area. It is the bargain that the patent system was designed to strike: a temporary monopoly in exchange for telling the world, in permanent and exhaustive detail, exactly how the thing works. The monopoly was the price the public paid. The disclosure is what the public got. And the disclosure outlives the monopoly forever.

Almost nobody talks about this. Walk into any bookshop and the machine-learning shelf will teach you what an algorithm does and how to code it, usually on top of a tower of dependencies that take a fast connection and a patient afternoon to install. None of those books will tell you who invented the technique, or when, or that the original description is sitting on a public website right now, free to read, written by the people who built it. None of them will tell you that the monopoly is over.

It matters that nobody is saying it, because we know what happens when somebody does. Patent expiry has quietly unlocked industries before. When the key desktop 3D-printing patents expired around 2009, the maker boom followed within months — the machines had been unaffordable not because they were hard to build, but because the right to build them was owned. When a drug patent expires, the generics industry turns a monopoly-priced molecule into medicine anyone can afford, the same chemistry at a fraction of the cost. Expiry is not a footnote in the history of technology; it is one of its engines. Machine learning's patent cliff — the great expiry of the filings from 1995 to 2006 — is the largest such unlocking in decades, and it has gone almost entirely unharvested. This book, and the library that accompanies it, is the harvest: the first attempt to do for machine learning what the generics industry did for medicine.

This book tells you. Each chapter takes one expired patent and does five things with it. It shows you the filing — the real document, its number, the company that owned it, the year, and how many later inventions cited it as prior art, which is the patent world's quiet measure of how much something mattered. It explains the idea from nothing, assuming you know no machine learning at all. It tells the story — who built it, what problem they were solving, and what the technique grew up to become. It gives you the code — a complete, working implementation you can run on a laptop, a ten-pound Raspberry Pi, or inside a web browser, with no library beyond the one Python ships with. And it closes with the inheritance: what this

newly-free algorithm is good for today, and where, surprisingly often, it still beats the heavyweight modern stack that replaced it.

The code is not pseudocode and it is not a sketch. It is the patentml library, which you can install in one line, and every listing in this book is lifted directly from its source. When you read a chapter and then run the algorithm, you are running the exact code on the page. The implementations are mine, released under the permissive MIT licence, so you may take them and do as you wish. The ideas are not mine and never were — they belong to the public domain, to the inheritance the patent system always promised would arrive. This book is a guided tour of that inheritance, on the morning after the vault unlocked.

Volume One is the half of the story concerned with prediction: the algorithms that search for a solution, and the algorithms that learn to decide. We begin with the strangest and oldest family of all — the optimisers that borrow their logic from evolution, from swarming birds, from the slow cooling of metal. We move to the neuron and the small, patient miracle of a network that learns from its own mistakes. And we arrive at the great workhorses of applied machine learning, the classifiers, including the single most-cited patent in this entire collection. By the end you will have built, from public-domain parts and nothing else, a complete system that takes in raw data and returns a decision.

Then, at the very end of Volume One, the ground will shift. Every algorithm in this volume assumes that somebody, somewhere, already knew the right answer and was willing to show it to the machine. Volume Two is about everything that happens when no one will.

Turn the page. The clock ran out years ago. It's all yours now.

HOW TO READ THIS BOOK

You need two things, and you almost certainly have both.

The first is high-school algebra. If you can read the expression $y = mx + b$ and have a rough memory of what a graph is, you have enough. Where a chapter needs a harder idea — a probability, a gradient, a vector — it builds that idea up from scratch, in plain language, at the moment you need it. There is no prerequisite chapter you were supposed to have read. There is no maths you are expected to already know. The few formulas that appear are there because a picture is sometimes the clearest sentence, never to impress you and never as a gate.

The second is Python, and only the standard kind. Install the companion library once:

`pip install patentml`

That is the entire setup. There is no `pip install numpy`, no compiler, no GPU, no account, no API key, no virtual environment you must nurse to health. The whole library is pure Python from the standard distribution. It will run on anything that runs Python 3.8 or later — which today includes locked-down work laptops, serverless functions, web browsers (through a project called Pyodide), and, with a little trimming, microcontrollers smaller than a postage stamp. The deliberate poverty of the dependencies is a feature, and Volume One's closing chapters explain why it is sometimes a decisive one.

Each chapter is self-contained. You may read this book front to back and watch the ideas build on one another, or you may open it to the algorithm you need this afternoon and ignore the rest. Both work. The code listings are complete — you can type or copy any of them and they will run — and because they come straight from the installed library, you can also simply import them:

```
from patentml import GeneticAlgorithm, RandomForest,  
ThompsonSampling
```

When you see a listing in the text, that is the real thing. Run it. Change the numbers. Break it and fix it. The whole point of an algorithm escaping into the public domain is that you are finally allowed to.

A NOTE ON PATENTS AND THE PUBLIC DOMAIN

A few honest words on the legal ground under this book, because the whole premise rests on it and you deserve to understand it rather than take it on faith. This is a plain-English map, not legal advice; if you are about to build a business on a specific patent, talk to a real attorney. But the shape of the thing is simpler than its reputation.

A patent is a bargain between an inventor and the public. The inventor describes an invention in complete, working detail and publishes that description for the whole world to read. In exchange, the government grants the inventor a temporary right to stop others from making or selling the invention — for a US utility patent filed in this era, twenty years from the filing date. That is the entire deal. It is not secrecy; it is the opposite of secrecy. A patent is a publication. The monopoly is the fee the public pays to be told how the thing works.

When the twenty years end, the monopoly ends with them, completely and permanently. The invention enters the public domain, where it stays forever. Anyone may now build it, use it, sell it, improve on it, or teach it, with no permission and no payment. A patent cannot be renewed back into force. Once it expires, it is everyone's, for good. Every patent discussed in this book was filed long enough ago that this has already happened. The dates are in the appendix; you can verify any of them yourself on Google Patents in about a minute, and the final appendix shows you how.

It is worth being precise about one distinction, because it is the hinge of the whole story. A patent is the opposite of a trade secret. A trade secret — the formula for a soft drink is the famous example — is protected only so long as it stays hidden, and it can in principle stay hidden forever. A patent protects something published, and only for twenty years. The institutions in this book made a choice, decades ago, to patent rather than to hide. That choice bought them two decades of monopoly. It also guaranteed that, when those two decades ran out, the knowledge would belong to all of us. They told us how, in writing, in exchange for time. The time is spent. The writing remains.

One last clarification, so there is no confusion as you read. The ideas in this book — the algorithms, the methods, the techniques described in the patents — are in the public domain and belong to everyone. The code in this book, the `patentml` library that implements those

ideas, is original work that I wrote and released under the MIT licence. You are free to use that code too, generously and commercially; the MIT licence asks almost nothing of you. But the two freedoms come from different places. The freedom to use the idea is a gift from the patent system and the passage of time. The freedom to use the code is a gift from me. Take both.

Now, to work.

PART I — EVOLUTION: ALGORITHMS THAT BREED SOLUTIONS

Before machine learning learned anything, it bred. The oldest family of techniques in this book does not reason its way to an answer and does not calculate one. It starts with a population of bad guesses, lets the better guesses have children, sprinkles in a little random mutation, and waits. Do this for long enough and the population fills with good answers — not because anything understood the problem, but because the bad answers stopped reproducing. It is the single strangest idea in machine learning, it predates almost everything else, and it works. This part of the book is its story, in six chapters. We begin with the purest form: the genetic algorithm itself.

CHAPTER 1

THE MACHINE THAT BRED ANSWERS (GENETIC ALGORITHMS)

THE FILING

US Patent 5,970,487 — "Genetic algorithm machine and its production method, and method for executing a genetic algorithm"
Assignee: Mitsubishi Electric Corporation. Filed 1997. Now expired; public domain.

US Patent 6,912,587 — "Method for utilizing a genetic algorithm to provide constraint-based routing of packets in a communications network"
Assignee: AT&T Corporation. Filed 2001. Now expired; public domain.

The first of these patents is a remarkable document to read, because Mitsubishi did not patent a piece of software. They patented a

machine. The claims describe physical hardware — a "population memory" to hold the candidate solutions, "chromosome registers" to load them into, a "crossover module" to mate them, a "mutation operator" to perturb them, and a "survival comparator" to decide which ones live. They were building the logic of biological evolution out of silicon and wire. The patent's quiet ambition is that this machine is not problem-specific: feed it any problem you can phrase as a fitness score, and the same hardware will evolve a solution. That generality is the whole idea, and it is why a single chip-level patent from 1997 still describes the technique we use in software today.

The AT&T patent, filed four years later, adds the piece that makes genetic algorithms genuinely useful in the messy real world. Its claims describe computing fitness "as a weighted sum of constraint fitness functions" — each requirement you care about gets scored separately, and then blended with a weight that says how much it matters. AT&T's particular problem was routing packets across a network under competing constraints (speed here, cost there, reliability everywhere), but the pattern is universal. Almost every real optimisation problem is really several problems at once, pulling in different directions. This patent is how you tell the machine how to weigh them.

THE IDEA

Suppose you want to find the best solution to a problem, and "best"

is something you can measure with a number — call that number the fitness. Higher fitness, better solution. You do not know how to compute the best solution directly; if you did, you would just compute it. But you can take any candidate solution and score it. That is all a genetic algorithm needs.

Here is the entire method, in the order it runs:

1. Start with a crowd. Generate a population of random candidate solutions. They will mostly be terrible. That is fine.
2. Score everyone. Run each candidate through the fitness function and give it its number.
3. Choose parents, with a thumb on the scale. Pick candidates to reproduce, but pick the fitter ones more often. A common way — the way the Mitsubishi "survival comparator" works — is a tournament: grab a handful at random, keep the best of that handful, repeat. Fit candidates win tournaments more often, so they get to be parents more often, but luck still gives the weak ones an occasional turn, which keeps the population from collapsing into one idea too early.
4. Mate them. Take two parents and combine their solutions to make children. The simplest blend is single-point crossover: pick a spot, take the first half of one parent and the second half of the other. The hope is that two solutions that are each good at different things will produce a child good at both.
5. Mutate, a little. With small probability, randomly tweak a child. Mutation is the source of genuinely new material — it lets the

population stumble onto ideas that neither parent had.

6. Replace, and repeat. The children become the new population. Go back to step 2. Do this for many generations.

That is the whole algorithm. Nothing in it understands your problem. Nothing in it reasons. It simply ensures, generation after generation, that good solutions have more children than bad ones — and that, plus a trickle of mutation, is enough to climb toward answers that no one designed. There is one detail worth protecting, and the patents both include it: elitism, the rule that you always carry the very best solution found so far, unchanged, into the next generation. Without it, a careless crossover can destroy your champion and you can actually get worse over time. With it, you can never go backwards.

THE STORY

The genetic algorithm is older than it looks. The core idea was worked out by John Holland at the University of Michigan in the 1960s and 70s, decades before the patents in this chapter. So what were Mitsubishi and AT&T patenting in the late 1990s, if the idea was already thirty years old?

They were patenting the machinery of applying it — and that distinction is the texture of this entire book. By the late 1990s, evolutionary computation had crossed from academic curiosity to industrial tool, and the industrial players raced to stake claims on the

practical apparatus: Mitsubishi on the hardware that could run evolution at silicon speed, AT&T on the specific, valuable trick of evolving solutions under many weighted constraints at once. These were not the inventors of evolution-as-computation. They were the engineers who industrialised it, and the patent record is where industrialisation leaves its fingerprints.

What did it grow into? Genetic algorithms and their cousins now quietly design things that are too tangled for a human to optimise by hand: antenna shapes for spacecraft (NASA famously flew an evolved antenna that looks like a bent paperclip and outperforms anything a person drew), delivery routes, factory schedules, the trading strategies in Volume Two's sister disciplines, even the architectures of other machine-learning models — a technique you will meet in Chapter 5, where evolution designs neural networks. The genetic algorithm is rarely the flashiest tool in the box, but it has a rare and valuable property: it asks almost nothing of your problem. It does not need your problem to be smooth, or differentiable, or convex, or even sensible. It only needs you to be able to score a guess. When everything else fails because the problem is too weird, the genetic algorithm often just works.

THE CODE

Here is the heart of the implementation from `patentml` — the main evolution loop, exactly as it ships. Read it against the six steps

above; you will find them one for one.

```
def evolve(self, init_population, fitness_fn, constraint_fns=None,
callback=None):
    population = init_population

    # Evaluate the initial population
    for chrom in population:
        if constraint_fns:
            self.weighted_constraint_fitness(chrom, constraint_fns)
        else:
            chrom.fitness = fitness_fn(chrom.genes)

    best = max(population, key=lambda c: c.fitness).clone()

    for gen in range(self.max_gens):
        # Elitism — carry the best N unchanged (US5970487 survival
        comparator)
        sorted_pop = sorted(population, key=lambda c: c.fitness,
reverse=True)
        new_pop = [c.clone() for c in sorted_pop[:self.elite_n]]

    # Fill the rest via selection + crossover + mutation
    while len(new_pop) < self.pop_size:
        p1 = self.tournament_select(population)
        p2 = self.tournament_select(population)
```

```
        c1, c2 = self.single_point_crossover(p1, p2)
        self.bit_flip_mutation(c1)
        self.bit_flip_mutation(c2)
        new_pop.extend([c1, c2])

population = new_pop[:self.pop_size]

# Score the new generation
for chrom in population:
    if constraint_fns:
        self.weighted_constraint_fitness(chrom, constraint_fns)
    else:
        chrom.fitness = fitness_fn(chrom.genes)

gen_best = max(population, key=lambda c: c.fitness)
if gen_best.fitness > best.fitness:
    best = gen_best.clone()

if callback:
    callback(gen, best)

return best
```

Notice how faithfully the code mirrors the patents. The elitism step is annotated with US5970487 because that is the "survival comparator" — the rule that the fittest survive untouched. The

constraint_fns branch is the AT&T contribution: when you supply weighted constraints, fitness becomes a weighted sum, exactly as claim 1 of US6912587 describes. The tournament selection, the single-point crossover, the small-probability mutation — every biological metaphor from the idea section is here as a named method.

Now let us actually use it. The "hello world" of optimisation is to find the highest point of a simple hill. We will ask the algorithm to find the value of x , somewhere between 0 and 6, that maximises $(x - 3)^2$ — a curve whose peak is obviously at $x = 3$, check that the machine found the right answer.

```
from patentml import GeneticAlgorithm, make_population
```

```
A population of 40 candidates, each a single number in [0, 6]
population = make_population(size=40, n_genes=1, gene_min=0.0,
gene_max=6.0)
```

```
ga = GeneticAlgorithm(population_size=40, mutation_rate=0.05,
max_generations=40, random_seed=7)
```

```
best = ga.evolve(population, fitness_fn=lambda genes: -(genes[0] -
3.0) ** 2)
```

```
print(f"best x found: {best.genes[0]:.4f}")    # !' c
```

The machine has never been told that the answer is 3. It does not know what the function means. It started with forty random numbers between 0 and 6, scored them, bred the better ones, mutated a few, and repeated forty times — and converged on the peak. That is evolution doing arithmetic.

The real power shows when you swap that toy fitness function for something you genuinely cannot solve by hand: a delivery schedule scored by total distance, a layout scored by how little it wastes, a portfolio scored by return against risk. The loop does not change. Only the `fitness_fn` changes. That indifference to the problem is precisely what Mitsubishi meant by a machine that is "not problem-specific," and it is why this one short function is, in a real sense, a general-purpose problem solver.

THE INHERITANCE

Now that this is free, where does it earn its place?

Reach for a genetic algorithm when your problem is weird* — when the thing you are optimising is bumpy, full of dead-ends, made of discrete choices rather than smooth dials, or simply too tangled to differentiate. The gradient-based methods you will meet in Chapter 10 are faster and more elegant when they apply*, but they need the problem to be smooth, and the real world frequently refuses. The genetic algorithm does not care. If you can score a guess, it can

optimise.

Reach for it, too, when you have several goals at once and need to be honest about the trade-offs between them. The AT&T weighted-constraint pattern is a clean, auditable way to say "speed matters twice as much as cost here," and to change your mind later by changing a number. That transparency is rare and valuable.

And reach for it precisely because it is now dependency-free public-domain code. A genetic algorithm in `patentml` is a few hundred lines of standard Python. It will run inside a constrained corporate environment that forbids installing libraries, inside a serverless function with a tight cold-start budget, on an embedded controller optimising a real machine in real time, or in a browser tab. You are not importing a framework and inheriting its weight and its risks; you are running a small, readable, public-domain idea that Mitsubishi taught the world how to build in 1997, and that, twenty years later, belongs to you.

The next chapter takes this same evolutionary engine and points it at a stranger target. Instead of evolving numbers, we will evolve programs — letting the machine breed not the answer to a problem, but the formula that produces the answer.

CHAPTER 2

PROGRAMS THAT WRITE PROGRAMS (GENETIC PROGRAMMING)

THE FILING

US Patent 6,477,444 — "Method for the automated design of decentralized controllers using genetic programming"

Assignee: Fuji Xerox Co., Ltd. Filed 2000. Now expired; public domain.

In the last chapter, a genetic algorithm evolved a number — the value of x that sat at the top of a hill. This patent evolves something far more ambitious. Its claims describe "constructing a plurality of programs using a genetic operation," and a crossover that works by "obtaining a subtree from one program and substituting it for a subtree in another program." Read that twice. Fuji Xerox is not patenting a way to find the best setting for a known formula. They are patenting a way to breed the formula itself — to evolve a

working program when you do not know, and could not write, the program you need.

The application that drove the patent was the automated design of controllers — the little decision-making programs that run a photocopier's mechanics, or a robot's limbs, or any decentralised system where many small parts must coordinate without a central brain. Writing such a controller by hand is hard precisely because the behaviour you want is easy to describe ("keep the paper feeding smoothly") and brutally hard to specify as code. The patent's answer is to stop writing the code at all. Describe what good behaviour looks like as a fitness score, and let evolution assemble a program that scores well. The claims lay out the loop: build random programs, score each one, keep and breed the high scorers, repeat.

THE IDEA

To evolve a program, you first have to store a program as something you can cut apart and recombine — something with the structural equivalent of genes. Genetic programming represents a program as a tree.

Picture the expression $x^2 + x$. You can draw it as a $+$ node; its left branch is a "square" node, whose own child is the variable x ; its right branch is simply the variable x . In this style is a tree like that. The branching nodes are operations —

add, multiply, divide, sine, square root, maximum. The tips, the leaves, are terminals — either an input variable constant number. To run the program on some input, you start at the leaves and work upward, each node combining its children's values, until the topmost node hands you a single answer.

Once a program is a tree, the moves from Chapter 1 all return, but they now operate on tree structure:

- A random program is a randomly grown tree: pick an operation for the root, then recursively grow random children, stopping at random leaves, until you hit a depth limit.
- Crossover — the move at the centre of the Fuji Xerox claim — takes two parent trees, snips a random branch (a "subtree") out of each, and swaps them. A useful fragment of logic from one program gets transplanted into another. This is how good sub-ideas spread through the population, not just good whole-programs.
- Mutation snips a random branch out of one tree and grows a fresh random branch in its place — the source of brand-new structure.
- Selection keeps the high-fitness trees as parents, exactly as before, with elitism preserving the champion untouched.

The most common use of this machinery is symbolic regression: you have a table of inputs and outputs, and you want a formula that reproduces them. Ordinary curve-fitting makes you choose the shape of the formula in advance (a straight line, a parabola) and only tunes

its coefficients. Symbolic regression chooses the shape too. It can hand you back $x^2 + x$ when that is what the data is. It finds an actual readable equation, discovered rather than assumed. There is one discipline the method needs: left unchecked, evolved trees tend to bloat into vast tangled monsters that fit the data by brute accident. So fitness usually carries a small size penalty — a gentle tax on complexity that nudges evolution toward formulas that are not just accurate but compact. This is Occam's razor, written as a line of code.

THE STORY

Genetic programming as a named discipline belongs largely to John Koza, who in the late 1980s and 1990s argued, against considerable scepticism, that evolution could produce not just optimised parameters but genuinely inventive structure — circuits, controllers, even patentable designs. The striking claim Koza pressed was that GP could be "an automated invention machine," producing results competitive with human engineering. The Fuji Xerox patent sits squarely in that tradition, applying the idea to the unglamorous but enormously valuable problem of designing controllers that no one wanted to hand-code.

What did it become? Symbolic regression — the formula-discovery use of GP — has had a quiet renaissance in the sciences, where researchers increasingly want not a black-box model that predicts

well but an actual equation they can read, interrogate, and reason about. A neural network can tell you that two quantities are related; a symbolic-regression formula can tell you how, in the universal language of mathematics. Evolved expressions have rediscovered physical laws from raw data and proposed relationships in materials science, biology, and finance. The appeal is the same one that drove Fuji Xerox: when you cannot write the program yourself, and you also refuse to settle for a model you cannot understand, you let evolution write you something legible.

THE CODE

The representation is the heart of it, so the implementation begins by defining what a program node is. Here, lightly condensed, are the tree node types from `patentml` — a variable, a constant, and a two-input operation:

```
class Var(GPNode):
    def __init__(self, idx): self.idx = idx
    def eval(self, x):      return float(x[self.idx]) # read input variable
    def __str__(self):      return f"x{self.idx}"

class Const(GPNode):
    def __init__(self, v):  self.v = round(float(v), 4)
    def eval(self, x):      return self.v             # a fixed number
```

```
class BinOp(GPNode):
    OPS = {"+": lambda a, b: a + b, "-": lambda a, b: a - b,
           "'": lambda a, b: a b,
           "/": lambda a, b: a / b if abs(b) > 1e-9 else 0.0, # protected
           "max": max, "min": min}
    divide
    def __init__(self, op, left, right):
        self.op, self.left, self.right = op, left, right
    def eval(self, x):
        return self.OPS[self.op](self.left.eval(x), self.right.eval(x))
```

Two small but important details of robustness are visible here. Division is protected: dividing by (near) zero returns zero instead of crashing, because an evolved program will absolutely try to divide by zero and you cannot have your optimiser die every time it does. And evaluation is wrapped to clamp wild values into a sane range — evolution is a chaotic process and the implementation has to survive its experiments.

The genetic operations work directly on this tree structure. Crossover is the Fuji Xerox claim, almost word for word:

```
def _crossover(self, a, b):
    """US6477444: obtain a subtree from one program, substitute into
    another."""
    child = a.clone()
```

```

nodes_c = collect_nodes(child)    # every node in the first parent
nodes_b = collect_nodes(b)        # every node in the second
parent
    tc = random.choice(nodes_c[1:]) # a random branch of the child
    tb = random.choice(nodes_b[1:]) # a random branch of the
donor
    return replace_node(child, id(tc), tb.clone()) # graft donor branch
in

```

And the evolution loop is the now-familiar rhythm — score, keep the elite, fill the rest with crossover, mutation, and a little fresh random blood, repeat — with the size penalty quietly taxing complexity at every evaluation:

```
fits = [fitness_fn(t) - self.size_pen * t.size() for t in pop]
```

Let us put it to work on the task the module itself uses as its demonstration: rediscovering a formula from nothing but a table of its inputs and outputs. We generate a hundred rows of the relationship $y = x^2 + x$, hand the machine only ask it for the formula.

```

import random, statistics
from patentml import GeneticProgramming

X = [[random.uniform(-3, 3), random.uniform(-3, 3)] for _ in

```

```
range(100)]
y = [row[0] ** 2 + row[1] for row in X]          # the hidden relationship

def fitness(tree):
    preds = [tree.eval(row) for row in X]
    mse = statistics.mean([(p - t) ** 2 for p, t in zip(preds, y)])
    return -mse                                # we maximise, so negate the
error

gp = GeneticProgramming(n_vars=2, pop_size=200,
n_generations=30,
                        random_seed=42, size_penalty=0.001)
best = gp.evolve(fitness)

print("discovered formula:", best)              # e.g. (sq(x0) + x1)
print("target was:      x0^2 + x1")
```

The machine is never told the answer. It is handed two hundred random trees, a way to score them, and thirty generations. It breeds expressions, transplants branches between them, mutates a few, taxes the bloated ones, and converges on a small tree that — when you `print it` — reads as some arrangement of $x \in \text{square}$ not fit coefficients to a formula you chose. It chose the formula.

THE INHERITANCE

Now that this is public-domain code, its sharpest use is the one the sciences have rediscovered: when you want an answer you can read. Reach for genetic programming when a black-box prediction is not enough — when you need the actual relationship, as an equation, because you intend to understand it, defend it, or build theory on top of it. A symbolic-regression formula is auditable in a way no neural network is. You can show it to a domain expert. You can check it against first principles. You can publish it.

Reach for it, too, when the shape of the solution is genuinely unknown. Conventional modelling makes you commit to a functional form up front; genetic programming discovers the form. That makes it a tool of first resort in the early, exploratory phase of a problem, when you do not yet know whether the relationship is linear, periodic, multiplicative, or something stranger — and you would like the data to tell you rather than guess.

And as with every algorithm in this book, it now carries no baggage. The patentml genetic programmer is a few hundred lines of standard Python with a tree, a fitness function, and a loop. It runs where heavyweight symbolic-regression packages cannot, it can be read end to end in an afternoon, and the technique at its core — breeding programs by swapping their branches — is Fuji Xerox's 2000 disclosure, now expired, now yours to point at any table of numbers that is hiding a formula.

The next two chapters round out the evolutionary family with optimisers that borrow their logic from elsewhere in nature — from the swarming of birds and fish, and from the slow, patient cooling of hot metal.

CHAPTER 3

THE SWARM (PARTICLE SWARM & DIFFERENTIAL EVOLUTION)

THE FILING

US Patent 6,212,427 — particle swarm optimisation.

Inventors: James Kennedy and Russell Eberhart — the two people who invented the technique. Filed 1999. Now expired; public domain.

US Patent 6,484,115 — differential evolution.

Inventors: Rainer Storn and Kenneth Price — again, the actual originators. Filed 1999. Now expired; public domain.

These two patents are unusual in this book because the names on them are the names in the textbooks. Most of the patents we examine were filed by corporations industrialising an academic idea; here, the inventors of the method and the holders of the patent are one and the

same. Kennedy and Eberhart described particle swarm optimisation in 1995, inspired by the choreography of bird flocks and fish schools, and patented the practical apparatus a few years later. Storn and Price introduced differential evolution in 1997 as a brutally simple, brutally effective optimiser for continuous problems. Both patents claim what their authors discovered, and both have now expired into the public domain. The core of the swarm patent is stated almost poetically in its claims: each particle "adjusts its position based on personal best and global best." That single sentence is the whole algorithm.

THE IDEA

Imagine a swarm of insects searching a dark landscape for the lowest point. Each insect remembers the best spot it personally has found so far. The swarm as a whole also broadcasts the best spot anyone has found. On every tick of the clock, each insect updates its velocity by being tugged in two directions at once: a little toward its own best memory, and a little toward the swarm's shared best. It keeps some of its existing momentum, adds those two tugs, and moves. That is particle swarm optimisation in full.

The balance between the two tugs is the art. The pull toward personal best is exploration — it keeps each particle investigating its own neighbourhood. The pull toward global best is exploitation — it draws the swarm together toward the best-known answer. Too much

social pull and the swarm collapses prematurely onto the first decent spot it finds; too much personal pull and it never converges. A common refinement, present in the implementation here, is inertia decay: early on, particles carry lots of momentum and roam widely; as the search proceeds, the inertia winds down and the swarm settles. Explore boldly at first, refine carefully at the end.

Differential evolution is a cousin with a different, almost startlingly simple trick. To improve a candidate solution, it does not compute a gradient or remember a best. It picks three other random members of the population, takes the difference between two of them, scales it, and adds it to the third to form a "donor." Then it blends the donor with the original candidate and keeps whichever scores better. The genius is that the difference between two existing solutions automatically provides a sensible step size and direction — when the population is spread out, steps are large; as it converges, the differences shrink and steps become fine. The algorithm tunes its own scale, for free, just by reading its own population.

THE STORY

Particle swarm optimisation came from an unexpected place: James Kennedy was a social psychologist, and the algorithm grew out of simulations of social behaviour — birds flocking, crowds aligning — rather than out of engineering. The insight that a simple rule of "follow your own best instinct and the crowd's best instinct" could

solve hard optimisation problems was genuinely surprising, and PSO became popular precisely because it is so easy to implement and so forgiving of messy problems. It has been used to tune everything from antenna arrays to neural-network weights to power-grid dispatch.

Differential evolution, from Storn and Price, won its reputation by winning competitions. In the optimisation contests of the late 1990s and 2000s, where dozens of methods were pitted against benchmark problems, DE repeatedly placed at or near the top despite — or because of — its simplicity. It became a default workhorse for continuous global optimisation, the method you reach for when you have a hard real-valued problem and want something that just works without much tuning. Both algorithms belong to the family of population-based optimisers that the genetic algorithm opened in Chapter 1; they differ only in how a population improves itself from one generation to the next.

THE CODE

The whole of particle swarm optimisation lives in one update rule, and patentml writes it exactly as Kennedy and Eberhart's patent claims it. Here is the velocity update — the line annotated as the core of US6212427 — surrounded by its position update:

Velocity update: US6212427 core equation

```
for d in range(n_dim):
    V[i][d] = (w * V[i][d]                      # keep some momentum
(inertia)
                + self.c1 * r1[d] * (pbest[i][d] - X[i][d]) # pull toward
personal best
                + self.c2 * r2[d] * (gbest[d] - X[i][d])) # pull toward global
best
    V[i][d] = max(-v_max[d], min(v_max[d], V[i][d])) # cap the
speed

Position update
for d in range(n_dim):
    X[i][d] += V[i][d]
    lo, hi = bounds[d]
    X[i][d] = max(lo, min(hi, X[i][d])) # stay inside the
bounds
```

Read the three terms of the velocity line against the idea: $w * V$ is retained momentum, the $c1$ term is the tug toward the particle's own best ($pbest$), and the $c2$ term is the tug toward the swarm's best ($gbest$). The two random vectors $r1$ and $r2$ make each tug jittery rather than mechanical, which keeps the swarm from marching in lockstep. The inertia weight w is decayed across iterations elsewhere in the loop — boldness early, precision late.

Using it is a matter of handing it a function to minimise and the

bounds of the search space:

```
from patentml import ParticleSwarmOptimiser
```

Minimise a 2-D bowl with its bottom at (3, -2)

```
def cost(p):
```

```
    return (p[0] - 3.0) ** 2 + (p[1] + 2.0) ** 2
```

```
pso = ParticleSwarmOptimiser(n_particles=30, n_iter=100,
```

```
random_seed=1)
```

```
best_point, best_cost = pso.minimise(cost, bounds=[(-10, 10), (-10, 10)])
```

```
print(f"found minimum at {best_point} (cost {best_cost})")  
near (3, -2)
```

Thirty particles, scattered at random across a twenty-by-twenty square, each following its own best instinct and the swarm's, converge on the bottom of the bowl in a hundred ticks. No gradients, no calculus — just a flock, tugged two ways.

THE INHERITANCE

Particle swarm and differential evolution earn their keep in the same territory as the genetic algorithm — problems too rough for calculus — but each brings a distinct temperament worth knowing. PSO is

fast, intuitive, and superb when you can picture the search as movement through a continuous space; it is a favourite for tuning the handful of real-valued knobs on an engineering design or a model. Differential evolution is the one to reach for when you want a continuous global optimiser that is robust almost to the point of being boring: few parameters, little tuning, strong results, the quiet competition-winner.

Their public-domain, dependency-free form makes them especially attractive as embedded optimisers — the kind of thing you drop into a control loop on a small device that must tune itself in the field, with no room for a heavyweight library. A swarm of thirty particles is a few arrays and a loop. It will run on a microcontroller. And because the inventors themselves placed these techniques on the public record and the twenty-year clock has run out, you may build that self-tuning device on Kennedy, Eberhart, Storn, and Price's own disclosures, owing nothing to anyone.

The next chapter turns to an optimiser with a completely different metaphor — not a crowd moving together, but a single point cooling slowly, like metal, into its lowest-energy state.

CHAPTER 4

ANNEALING THE ANSWER (SIMULATED ANNEALING & ANT COLONY OPTIMISATION)

THE FILING

US Patent 7,219,040 — "Method and system for model based control of heavy duty gas turbine and combined cycle power plants", embodying constraint-based optimisation.

Assignee: General Electric Company. Filed 2002. Now expired; public domain.

US Patent 6,668,201 — multi-objective controller tuning via fitness functions.

Assignee: General Electric Company. Filed 2000. Now expired; public domain.

Ant colony optimisation appears in this chapter as the swarm's stranger sibling; its lineage runs through the broad body of expired

evolutionary-computation patents, and `patentml` implements it alongside the neural code in its `deep_networks` module.

General Electric's patents here are not, on their face, about a famous algorithm. They are about running power plants and tuning controllers — deeply practical industrial problems. But buried in their claims is a general and valuable pattern: constraint-based optimisation, the art of finding a good solution while respecting hard limits ("never exceed this temperature," "keep emissions under that threshold"). The technique GE leaned on for the global search is simulated annealing, and what makes their disclosure useful to us is the way they handle constraints — by turning each violation into a penalty added to the cost. We will see that exact pattern in the code.

THE IDEA

Simulated annealing takes its name and its logic from metallurgy. To make metal strong, a smith heats it until its atoms jostle freely, then cools it slowly so they settle into a low-energy, well-ordered crystal. Cool it too fast and the atoms freeze into a brittle, flawed arrangement; cool it slowly and patiently and they find a near-perfect structure. Simulated annealing performs the same ritual on a mathematical problem.

You have a single candidate solution and a "cost" you want to minimise — think of cost as energy, lower is better. At each step you

propose a small random change, a neighbour. If the neighbour is better, you take it; that much is obvious. The crucial, counterintuitive move is what happens when the neighbour is worse: you sometimes take it anyway, with a probability that depends on two things — how much worse it is, and how "hot" the system currently is. Early on, the temperature is high and the algorithm cheerfully accepts bad moves, wandering widely and escaping the local dips that would trap a greedy method. As the temperature cools on a schedule, it grows pickier, accepting fewer and smaller backward steps, until near the end it only ever moves downhill, settling into the best basin it has found.

That willingness to go uphill sometimes is the entire point. A method that only ever improves will march straight into the nearest small valley and stop, mistaking it for the lowest point in the landscape. Simulated annealing pays a temporary price — accepting worse solutions while it is hot — to buy the ability to climb out of small valleys and discover deeper ones. The acceptance probability that governs this, $\exp(-E / T)$, is borrowed directly from physics: it tells how often a system at temperature T will spontaneously jump to a higher-energy state. It is thermodynamics, repurposed as a search strategy.

Ant colony optimisation, the chapter's companion, borrows from a different natural process: the way real ants find short paths. An ant wandering to food lays down a pheromone trail on the way back;

other ants tend to follow strong trails and reinforce them; shorter paths get traversed faster, so they accumulate pheromone faster, so they attract still more ants — a feedback loop that converges the colony on good routes. As a computer algorithm, this is a beautiful fit for problems about paths and orderings — the travelling-salesman problem, network routing, scheduling — where a solution is a sequence of choices and "pheromone" is a memory of which choices have worked well.

THE STORY

Simulated annealing was introduced in 1983 by Kirkpatrick, Gelatt, and Vecchi, who made the explicit and fruitful analogy between optimisation and the statistical mechanics of cooling solids. It became one of the most widely used global optimisers of the following decades precisely because it is simple, general, and comes with an intuition anyone can hold: heat it up, cool it down slowly, don't be greedy too early. General Electric's patents show the technique doing exactly the unglamorous, high-stakes work it excels at — tuning the controllers of gas turbines and power plants, where a better configuration is worth real money and the constraints are not optional.

Ant colony optimisation came a decade later, from Marco Dorigo's 1992 doctoral work, and found its home in routing and logistics — the problems where the answer is a path and the search space is the

dizzying number of ways to order a set of stops. Both algorithms are part of the same broad rebellion that runs through this part of the book: the discovery, again and again, that processes borrowed from nature — evolution, flocking, cooling, foraging — are startlingly good general-purpose problem solvers, and that you do not need to understand a problem deeply to optimise it well, only to imitate the right natural process.

THE CODE

The heart of simulated annealing is the acceptance rule, and `patentml` writes it in the few lines that matter. Here is the inner loop — propose a neighbour, measure the change in cost, and decide whether to accept:

```
for t in range(self.n_steps):
    T = self._temperature(t)      # how hot are we right now?
    x_n = self._neighbour(x, bounds)  # propose a small random
change
    c_n = full_cost(x_n)
    dE = c_n - current_cost        # "E: position
# Accept if better, OR probabilistically if worse (the annealing
move)
    if dE < 0 or random.random() < math.exp(-dE / max(T, 1e-10)):
        x, current_cost = x_n, c_n
```

```

if c_n < best_cost:
    best_x, best_cost = list(x_n), c_n

```

Everything is in that if. When $dE < 0$ the neighbour is better and we always take it. Otherwise we take it with probability which is near 1 when the temperature T is high (accept almost anything) and shrinks toward 0 as T cools (accept almost nothing that is worse). The temperature itself follows a cooling schedule; the implementation offers exponential, linear, and adaptive cooling, the exponential one simply multiplying the temperature down toward its floor over the run.

GE's constraint-handling pattern is the other thing worth seeing, because it is so reusable. Constraints are not enforced by forbidding moves; they are folded into the cost as squared penalties:

```

def full_cost(sol):
    cost = obj_fn(sol)
    if constraint_fns:
        for cfn in constraint_fns:
            violation = max(0.0, cfn(sol))    # how badly is this rule
            broken?
            cost += self.cw * violation ** 2  # pay a steep, growing penalty
    return cost

```

A solution that respects every constraint pays no penalty; one that

violates a rule pays a price that grows with the square of the violation, so small breaches are tolerated while large ones become unbearable. The optimiser is then free to search the whole space, including briefly-illegal regions, while being relentlessly pushed back toward legality. That is the GE disclosure in five lines.

Putting annealing to work looks like every optimiser in this part — a function to minimise and some bounds:

```
from patentml import SimulatedAnnealing
```

```
def cost(p):
```

```
    return (p[0] - 3.0) ** 2 + (p[1] + 2.0) ** 2
```

```
sa = SimulatedAnnealing(T_start=100.0, T_end=0.01, n_steps=5000,  
random_seed=1)
```

```
best_point, best_cost = sa.minimise(cost, bounds=[(-10, 10), (-10,  
10)])
```

```
print(f"annealed to {best_point} (cost {best_cost})  
(3, -2)
```

A single point, heated and cooled over five thousand steps, wanders the landscape boldly while hot and settles precisely while cold — and lands in the bottom of the bowl.

THE INHERITANCE

Simulated annealing is the optimiser to reach for when your landscape is riddled with local traps* and you are willing to trade speed for the confidence of a thorough, trap-escaping search. It shines on combinatorial problems — layouts, schedules, assignments — where the solutions are discrete and the terrain is jagged, and it is famously easy to apply: if you can measure a cost and propose a small change, you can anneal. GE's squared-penalty trick travels with it and is worth stealing on its own, a clean and general way to bolt hard constraints onto any* optimiser in this book.

Ant colony optimisation is the specialist for paths and orderings — routing fleets, sequencing jobs, threading networks — the problems where a solution is a route and the search is over the staggering number of ways to arrange the stops.

Both are now public-domain, dependency-free, and small. A simulated annealer is a loop, a cost function, and a random number; it will run inside a control system tuning itself in the field, inside a serverless function solving a layout on demand, or inside a browser. General Electric used these techniques to tune power plants two decades ago and told the world how in patents that have now expired. The cooling schedule, the acceptance rule, the penalty pattern — all of it is yours.

The next two chapters bring the evolutionary part of the book to its most self-referential point. First we let evolution design neural networks; then we build optimisers that learn a model of where good solutions hide and use it to search more cleverly than blind variation ever could.

CHAPTER 5

EVOLVING THE NETWORK ITSELF (NEUROEVOLUTION)

THE FILING

US Patent 5,835,901 — "Perceptive system including a neural network" with parameters adjusted by evolutionary techniques. Assignee: Lockheed Martin Corporation. Filed 1997. 142 forward citations. Now expired; public domain.

US Patent 7,162,461 — a hybrid neural network whose activation function is "selected from a candidate set," with multiple configurations evaluated. Assignee: SAS Institute. Filed 2005. Now expired; public domain.

This chapter is where the evolutionary part of the book folds back on itself. The previous four chapters used evolution to optimise numbers and programs. Here, evolution optimises a neural network

— the very thing that the next part of the book trains by gradient descent. Lockheed's patent, with its heavy citation count signalling real influence, describes adjusting a neural network's parameters "using evolutionary techniques" rather than the usual calculus. SAS's later patent adds a twist that gradient methods cannot easily manage: evolving the network's activation functions themselves, choosing from a candidate set, treating the shape of each neuron as something to be searched rather than fixed in advance.

That Lockheed Martin — an aerospace and defence company — holds the foundational patent is telling. Neuroevolution thrives exactly where gradient descent struggles: in control problems, in systems that must adapt in real time, in situations where you can measure how well a controller performs but cannot easily say what its correct output should have been at each instant. Those are the problems of guidance, robotics, and autonomy.

THE IDEA

A neural network, which the next part of this book treats in detail, is at heart a big bag of numbers — its weights — arranged so that inputs flow through layers of simple units and emerge as an output. Training a network means finding good values for those weights. The standard way, the subject of Chapters 7 through 10, is gradient descent: nudge each weight in the direction that reduces error, using calculus to compute the direction. It is fast and effective — when

you can compute the error gradient.

But sometimes you cannot. Imagine evolving the controller for a walking robot. You can let the controller run and measure how far the robot walked before falling — that is a fitness score. But you cannot say what the correct motor torque was at millisecond 4,217, so there is no per-step error to take a gradient of. The reward is sparse, delayed, and global. Gradient descent has nothing to bite on. Neuroevolution does.

The idea is simply to apply the machinery of the last four chapters to the network's weights. Treat the entire flattened list of weights as a "genome." Make a population of networks with random weights. Score each by letting it perform the task and measuring fitness. Keep the best — the elites — and breed the next generation by copying the winners and mutating their weights with small random perturbations. Repeat. The network never computes a gradient; it improves because better-performing weight-sets have more offspring. This particular flavour, keeping the top performers and filling the rest with mutated copies, is a classic evolutionary strategy, sometimes

The SAS contribution pushes further. Why fix every neuron to use the same activation function — the same little non-linear shape — when you could let evolution choose each one from a menu of candidates: relu, sigmoid, tanh, sine, and more? A gradient method cannot do this easily, because the choice of activation is a discrete

decision, not a smooth dial. Evolution does not care about smoothness. It can search over the architecture — the very structure and character of the network — as freely as over the weights.

THE STORY

Neuroevolution has a long and slightly underdog history. While the mainstream of neural-network research poured its energy into perfecting gradient descent — backpropagation, the subject of the next chapter, became the dominant paradigm — a persistent minority argued that evolution could train networks too, and could do things gradients could not: discover network topologies, optimise non-differentiable objectives, and handle the sparse-reward control problems where backprop founders. The most celebrated method in this tradition, NEAT (NeuroEvolution of Augmenting Topologies), grew networks from simple to complex by evolving both weights and structure, and produced striking results in game-playing and control.

Lockheed's 1997 patent, filed at the height of the first neural-network boom and cited 142 times since, planted an early industrial flag in this ground, aimed at "perceptive systems" that learn in real time — the language of sensing and control. The technique never displaced gradient descent for the big supervised problems of image and language, where labelled data and differentiable losses make calculus king. But it carved out a durable niche in reinforcement learning and

robotics, and it enjoyed a genuine resurgence when researchers showed that simple, massively-parallel evolutionary strategies could rival sophisticated gradient-based reinforcement learning on hard control tasks — a vindication of the idea Lockheed patented two decades earlier.

THE CODE

The implementation makes the central trick concrete: a network whose weights can be flattened into a vector, perturbed, and reloaded. Here is the EvoNet — a plain feed-forward network — exposing exactly the operations evolution needs: read out all the weights, write them back, and mutate them.

```
def get_params(self):
    """Flatten every weight and bias into one long vector — the
    'genome'."""
    params = []
    for W in self.weights:
        for row in W:
            params.extend(row)
    for b in self.biases:
        params.extend(b)
    return params

def mutate(self, sigma=0.1, prob=0.1):
```

```
""Gaussian mutation of weights — the evolutionary move.""
params = self.get_params()
for i in range(len(params)):
    if random.random() < probab:
        params[i] += random.gauss(0, sigma) # nudge a weight at
random
self.set_params(params)
```

That is the whole interface between a neural network and an evolutionary optimiser: a network is a vector you can read, write, and jiggle. The WeightEvolution engine wraps it in the now-familiar loop — score the population, keep the elites, fill the rest with mutated copies — annotated against Lockheed's claim about "adjusting network parameters using evolutionary techniques." Notice the candidate activation set, which is the SAS patent's contribution, sitting ready to be evolved:

```
ACTIVATIONS = {
    "relu": lambda x: max(0.0, x),
    "sigmoid": lambda x: 1.0 / (1.0 + math.exp(-clamp(x))),
    "tanh": lambda x: math.tanh(x),
    "linear": lambda x: x,
    "elu": lambda x: x if x >= 0 else math.exp(x) - 1,
    "swish": lambda x: x / (1.0 + math.exp(-clamp(x))),
    "step": lambda x: 1.0 if x > 0 else 0.0,
    "sin": lambda x: math.sin(x),
```

```
}
```

To use it, you build a template network, wrap it in the evolver, and supply a fitness function that scores a network by what it does:

```
from patentml import EvoNet, WeightEvolution

template = EvoNet(layer_sizes=[2, 6, 1])    # 2 inputs, a hidden
layer, 1 output

def fitness(net):
    # however well the network performs the task — higher is better
    return -task_error(net)

evolver = WeightEvolution(template, pop_size=20, n_elites=5,
random_seed=1)
best_net = evolver.evolve(fitness, generations=100)
```

The crucial line is the fitness function. It does not need to know what the network should have output at each step — only how well the whole network performed. That is the freedom neuroevolution buys: it optimises behaviour you can score but cannot differentiate.

THE INHERITANCE

Reach for neuroevolution when you can measure success but cannot

compute a gradient — the defining situation of control and reinforcement learning. A robot that walks, a game-playing agent rewarded only at the end, a controller judged by its overall behaviour over time: these have fitness but no clean per-step error, and that is precisely where gradient descent has nothing to grip and evolution thrives. Reach for it, too, when you want to search over architecture — the choice of activations, the shape of the network, the discrete structural decisions that calculus cannot touch.

There is a practical bonus that the public-domain, dependency-free form makes vivid. Evolutionary training is embarrassingly parallel and needs no automatic-differentiation machinery — no heavyweight framework computing gradients through a computation graph. Each network in the population is scored independently; you can spread them across cores or machines trivially. And the whole apparatus, in `patentml`, is a network you can flatten and a loop that mutates it — small enough to run on the very devices that neuroevolved controllers are meant to control. Lockheed built this for perceptive systems that learn in the field in 1997. The patent has expired. The field is now yours.

The last chapter of this evolutionary part takes one more step in sophistication. Instead of improving a population by blind variation, we will build optimisers that learn a model of where good solutions live — and use that model to generate the next generation deliberately, not by luck.

CHAPTER 6

GUESSING SMARTER (BAYESIAN OPTIMISATION & ESTIMATION OF DISTRIBUTION)

THE FILING

US Patent 7,047,169 — "Method for optimizing a solution set" using a probabilistic selection operator.

Assignee: University of Illinois. Filed 2002. Now expired; public domain.

The claims of this patent describe a loop that is subtly but profoundly different from everything earlier in this part of the book. Instead of breeding solutions by crossover and mutation, it works like this: "generating a solution set using a probabilistic model," "selecting a subset with high fitness," "estimating a new probabilistic model from the selected subset," and "sampling new solutions from the new model." Read that sequence carefully. There is no crossover here, no mutation. The algorithm builds a statistical model of what

good solutions look like, then samples the next generation directly from that model. This family is called the Estimation of Distribution Algorithm (EDA), and it is the bridge from blind evolution to the modern, model-guided optimisation that powers automated machine-learning tuning today.

The University of Illinois was a natural home for this work — it was the long-time base of genetic-algorithm research, and the EDA idea grew directly out of asking a sharp question about ordinary genetic algorithms: what are crossover and mutation actually trying to do? The answer, the patent's authors realised, is that they are clumsily trying to recombine the good building blocks of fit solutions. So why not model those building blocks explicitly — learn the distribution of good solutions — and sample from it directly, skipping the clumsiness?

THE IDEA

Every optimiser in this part so far has improved a population by perturbing its members: mutate a weight, swap a subtree, nudge a particle. The estimation-of-distribution idea throws that out and replaces it with two repeating steps:

1. Learn what good looks like. Take the current population, keep the fittest fraction, and fit a probability distribution to them. In the simplest and most useful case, you model each dimension of the

problem as a bell curve — a Gaussian — and you compute its mean and spread from the selected winners. Now you have a statistical portrait of the good region: "the best solutions tend to have their first coordinate around 3.1, give or take 0.4; their second coordinate around 0.6, give or take 0.6."

2. Sample the next generation from that portrait. Instead of mutating individuals, you draw brand-new candidates straight from the distribution you just learned. They are not children of any particular parents; they are fresh samples from your current best understanding of where good solutions live.

Repeat, and something elegant happens. Early on, the selected winners are scattered, so the distribution is wide and the sampling explores broadly. As the search homes in, the winners cluster, the distribution narrows around them, and the sampling automatically becomes a fine local search. The algorithm's uncertainty — the width of its bell curves — shrinks as it learns, and that shrinking spread is the transition from exploration to exploitation, managed automatically and visibly. You can literally watch the standard deviations decrease and know the search is converging.

This is the conceptual seed of two important modern things. One is CMA-ES, a more sophisticated cousin that models not just per-dimension spread but the full correlated shape of the good region, and which remains a gold-standard optimiser for hard

continuous problems. The other is Bayesian optimisation in the sense now ubiquitous in machine learning: the method of choice for tuning the hyperparameters of expensive models, where each trial costs real time or money and you cannot afford blind search. Both descend from the same insight the Illinois patent crystallised — that you should build a model of where to look, and let it guide you, rather than searching at random.

THE STORY

The estimation-of-distribution family emerged in the late 1990s and early 2000s as researchers — many at Illinois, in the genetic-algorithms tradition founded there — sought to put evolutionary computation on firmer probabilistic footing. The motivating critique was honest and sharp: ordinary genetic algorithms are powerful but somewhat blind, recombining building blocks they never explicitly identify. EDAs made the building blocks explicit by modelling them, trading some of the genetic algorithm's cheerful simplicity for a more principled, statistically-grounded search.

The pay-off arrived when the cost of each evaluation became the bottleneck. In modern machine learning, testing one configuration of a model can mean hours of training; you cannot afford the thousands of evaluations a blind optimiser happily burns. Model-guided optimisation — learn from every trial, build a picture of the response

surface, and spend each precious evaluation where the model says it will help most — became indispensable. Today, automated hyperparameter tuning, automated machine learning, and experimental design in the sciences all lean on this idea. The Illinois patent, filed in 2002 and now expired, sits at the headwaters of that river: the moment evolutionary search learned to model rather than merely to vary.

THE CODE

The two-step rhythm is the whole thing, and `patentml` writes each step plainly. First, learning the model — estimating a mean and spread per dimension from the selected winners, exactly as the patent's "estimating a probabilistic model from selected subset" requires:

```
def _estimate_model(self, selected):
    """US7047169: estimate a probabilistic model from the selected
    subset."""
    n = len(selected)
    for d in range(self.n_vars):
        vals = [ind[d] for ind in selected]
        mu = sum(vals) / n                # mean of the winners
        var = sum((v - mu) ** 2 for v in vals) / max(n - 1, 1)
        self.means[d] = mu
        self.sigmas[d] = math.sqrt(var)   # spread of the
```

winners

Second, sampling the next generation directly from that model — no crossover, no mutation, just fresh draws from the learned bell curves:

```
def _sample(self):  
    return [random.gauss(m, max(self.sigma_min, s))  
            for m, s in zip(self.means, self.sigmas)]
```

And the main loop strings them together in the order the patent claims them — sample a population from the current model, evaluate, select the best, re-estimate the model, repeat:

```
for gen in range(self.n_gens):  
    population = [self._sample() for _ in range(self.pop_size)] #  
sample from model  
    fits = [fitness_fn(ind) for ind in population]                # evaluate  
  
    ranked = sorted(zip(fits, population), key=lambda x: -x[0])  
    selected = [ind for _, ind in ranked[:self.select_n]]        # keep the  
best  
    self._estimate_model(selected)                                # learn a new  
model
```

In use, it looks like the other optimisers — but watch what you can observe through the callback:

```
from patentml import GaussianEDA

def fitness(p):
    return -((p[0] - 3.0) ** 2 + (p[1] + 2.0) ** 2)
(3, -2)

eda = GaussianEDA(n_vars=2, pop_size=100, n_generations=50,
random_seed=1)
best_point, best_fit = eda.optimise(fitness, bounds=[(-10, 10), (-10,
10)])

print(f"converged on {best_point}")      # !' near
```

Behind that result, the two standard deviations the model tracks start near 1.0 and shrink toward zero as the generations pass. That shrinking is the algorithm becoming certain. Unlike the blind optimisers earlier in this part, this one carries an explicit, inspectable belief about where the answer is and how sure it is — and that belief is what makes it efficient with expensive evaluations.

THE INHERITANCE

Reach for estimation-of-distribution and its Bayesian-optimisation descendants when each evaluation is expensive and you cannot afford to be blind. This is the optimiser for tuning the settings of a slow model, for designing a costly physical experiment, for any

problem where a trial costs minutes, money, or material and you must extract maximum information from every one. By learning from all its past trials and modelling the landscape, it spends evaluations deliberately rather than scattering them — the decisive advantage when evaluations are scarce.

Reach for it, too, when you want an optimiser that knows what it knows. The explicit model of uncertainty is not just a means to an end; it is information you can read. A narrowing spread tells you the search is converging; a stubbornly wide one tells you the problem is hard or multi-modal and you should be suspicious of any single answer. Few optimisers are so willing to show you their own confidence.

And in its public-domain, dependency-free form, the basic estimation-of-distribution algorithm is astonishingly small — a few means, a few standard deviations, a sampler, and a loop. The University of Illinois disclosed this principled, model-guided search in 2002, the patent has expired, and you can now drop the seed of modern automated optimisation into a controller, a script, or a browser, owing no one anything.

That completes the evolutionary opening of the book. We have watched solutions bred, programs evolved, swarms converge, metal cool, networks evolve, and search learn to model itself — six optimisers, all borrowed from nature, all now free. Part II changes

register entirely. We leave the population behind and turn to the single most consequential idea in modern machine learning: the neuron, and the small, patient miracle of a network that learns from its own mistakes by calculus.

PART II — THE NEURON: HOW NETWORKS LEARNED TO LEARN

If Part I was machine learning's wild, biological childhood — breeding, swarming, cooling — Part II is its disciplined adolescence. Here we meet the single idea that would, decades after these patents were filed, come to dominate the entire field: the artificial neuron, stacked into layers, trained by calculus to learn from its own mistakes. The four chapters of this part build that idea up from one neuron to convolutional and recurrent networks, and end with the gradient-descent optimisers that make all of it learn. None of this is deep learning at the billion-parameter scale — it is the public-domain bedrock that the billion-parameter scale was built on, small enough to read and run, filed by AT&T, Microsoft, Texas A&M, and Lockheed before the world was watching.

CHAPTER 7

BACKPROP IN A BOX (THE MODULAR NEURAL NETWORK)

THE FILING

US Patent 6,128,606 — "Module for constructing trainable modular network in which each module outputs and inputs data structured as a graph"

Assignee: AT&T Corporation. Filed 1997. 53 forward citations. Now expired; public domain.

Behind this patent stands one of the most important research groups in the history of machine learning. The AT&T / Bell Labs team of the 1990s — the names most associated with it are Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner — built the trainable, modular, gradient-descent-based recognisers that read the handwritten digits on millions of bank cheques. Two of those researchers would, two decades later, share the Turing Award for the

deep-learning revolution. The patent's language is precise and prophetic: a network assembled from trainable modules, joined into a graph, optimised by "back-propagated gradients" against an "objective function." That is the architecture of essentially every neural network trained since. The famous machine that came out of this work, a convolutional network called LeNet, was reading cheques commercially while the rest of the world still thought of neural networks as an academic curiosity.

THE IDEA

Strip a neural network to its smallest piece and you have an artificial neuron: a unit that takes several numbers in, multiplies each by a weight, adds them up with a bias, and passes the total through a simple non-linear function — squashing it, say, to lie between 0 and 1, or clipping away the negatives. One neuron is almost nothing. The power comes from arranging many of them in layers, the outputs of one layer feeding the inputs of the next, so that data flows from the input, through one or more hidden layers, to an output. This is the feed-forward network, and running data through it — the forward pass — is just a cascade of those weighted sums and squashes.

The miracle is in the training, and the miracle has a name: backpropagation. A freshly built network has random weights and produces garbage. To improve it, you show it an example, let it produce its (wrong) output, and measure how wrong it is with a loss

— a single number, the error. Now the question is: which way should each weight move to make the loss smaller? There can be thousands of weights, and the answer for each depends on all the others, so it looks hopeless. Backpropagation makes it tractable. It is, at heart, the chain rule of calculus applied cleverly: compute the error at the output, then push it backward through the network layer by layer, working out how much each weight contributed to the error and therefore which direction it should move. One forward pass to make a prediction; one backward pass to assign blame. Then nudge every weight a little in its blame-reducing direction, and repeat over thousands of examples. The network descends, step by step, toward weights that work.

One more refinement makes it practical: rather than update the weights after every single example (noisy) or only after seeing all of them (slow), you process a small mini-batch at a time, average the gradients across it, and step on that average. This is the rhythm of nearly all neural-network training, and it is exactly what the AT&T patent describes — gradients, back-propagated, against an objective, with a learning rate controlling the size of each step.

THE STORY

In the 1990s this was a minority pursuit. Neural networks had been written off once already, in the 1970s, and the dominant machine-learning methods of the era were the support vector

machines and decision trees we will meet in Part III. But the AT&T / Bell Labs group kept building, and they had something most academics did not: a real, high-volume, money-on-the-line problem — reading the account numbers and amounts on handwritten cheques — and the engineering will to ship a neural network that solved it. The "modular" framing of the patent is the deep insight that would eventually win: build the network out of composable, trainable pieces, each able to pass gradients to its neighbours, and you can assemble arbitrarily complex architectures and train the whole thing end to end.

For years, this approach was admired by a few and ignored by many. Then, in the early 2010s, three things arrived at once — vast labelled datasets, the parallel horsepower of graphics cards (the subject of the next chapter, foreseen in a 2005 patent), and a few key refinements — and the modular, backprop-trained network suddenly outperformed everything else at image recognition by a stunning margin. The revolution that followed, the one that produced the systems now reshaping the world, is the direct descendant of the modest cheque-reader described in this AT&T patent. The idea did not change. The scale did.

THE CODE

The patentml neural network keeps the structure visible. The forward pass is exactly the cascade of weighted sums and squashes described

above:

```
def forward(self, x):
    acts = [x]
    cur = x
    for W, b in zip(self.weights, self.biases):
        # weighted sum into each neuron of the next layer, plus bias
        z = [sum(cur[i] * W[i][j] for i in range(len(cur))) + b[j]
              for j in range(len(b))]
        cur = [self.act(zi) for zi in z] # squash through the activation
        acts.append(cur)
    return cur, acts          # keep every layer's activations for the
backward pass
```

The training step is backpropagation made literal — a mini-batch version, exactly as the AT&T patent frames it. The error at the output is computed, then pushed backward through the layers, accumulating how much each weight is to blame:

```
delta = [(o - t) * 2 / len(y) for o, t in zip(out, y)] # error at the output

for l in range(len(self.weights) - 1, -1, -1):          # walk layers
backward
    prev_act = acts[l]
    for i in range(len(prev_act)):
        for j in range(len(delta)):
```

```

        grad_W[l][i][j] += prev_act[i] * delta[j]    # blame for each
weight
...
    if l > 0:                                     # propagate error further back
        delta = [sum(self.weights[l][i][j] * delta[j] for j in
range(len(delta)))
                * self.d_act(prev_act[i]) for i in range(len(prev_act))]

```

After accumulating gradients across the whole mini-batch, the weights step in the blame-reducing direction, scaled by the learning rate:

```

for l in range(len(self.weights)):
    for i in range(len(self.weights[l])):
        for j in range(len(self.weights[l][i])):
            self.weights[l][i][j] -= self.lr * grad_W[l][i][j] / n # gradient
descent

```

The classic proof that a network can learn something a single neuron cannot is the XOR problem — output 1 when exactly one of two inputs is 1. No straight line separates its cases, so no single neuron can solve it; you need a hidden layer. Here is the network learning it:

```

from patentml import MiniNeuralNet

```

```

X = [[0, 0], [0, 1], [1, 0], [1, 1]]

```

```

y = [[0], [1], [1], [0]]      # XOR — not linearly separable

net = MiniNeuralNet([2, 4, 1], learning_rate=0.1) # 2 in, 4 hidden,
1 out
for epoch in range(1000):
    net.train_batch(X, y)

for xi in X:
    out, _ = net.forward(xi)
    print(xi, "->", round(out[0]))      # !' 0, 1,

```

Started from random weights, the network sees the four XOR cases a thousand times, each pass nudging its weights down the error gradient, and learns a function that a single neuron provably cannot represent. The hidden layer is what makes it possible; backpropagation is what makes the hidden layer learnable.

THE INHERITANCE

The feed-forward network is the most general-purpose learner in this book: given enough hidden units and enough examples, it can approximate essentially any relationship between inputs and outputs. That universality is why it became the foundation of the field. Reach for it whenever you have a supervised problem — inputs paired with the answers you want — and a relationship too tangled for a straight line or a tree, and you are willing to train.

What the public-domain, dependency-free form offers is something the giant frameworks cannot: a neural network you can read in full. Every weight, every gradient, every line of the backward pass is in front of you, in plain Python, with nothing hidden inside a compiled library. For learning how learning works there is nothing better — you can put a print statement anywhere, watch the gradients flow, break it and see what breaks. And for genuinely small problems on genuinely small hardware — a handful of inputs, a modest hidden layer, an embedded controller that must adapt — the few hundred lines here will do the job where importing a deep-learning framework would be absurd. AT&T built the modular, backprop-trained network to read cheques in 1997. The patent is expired. The idea that conquered the field is now a file you can open.

The next two chapters specialise the neuron for the two great kinds of structured data — grids and sequences — with the convolutional and recurrent networks that AT&T, Microsoft, and Texas A&M patented before either was famous.

CHAPTER 8

THE GPU PROPHECY (CONVOLUTIONAL NEURAL NETWORKS ON GRAPHICS HARDWARE)

THE FILING

US Patent 7,747,070 — "Training convolutional neural networks on graphics processing units"

Assignee: Microsoft Corporation. Filed 2005. 91 forward citations. Now expired; public domain.

This may be the most quietly prophetic patent in the entire book. In 2005 — years before the deep-learning revolution, when neural networks were still a fringe interest — Microsoft filed a patent on training convolutional neural networks on graphics processing units. Read the date again. The single most important enabling fact of the modern AI era is that the massively parallel chips designed to draw video-game graphics turned out to be nearly perfect engines for training neural networks. That realisation, when it arrived in force

around 2012, changed everything. This patent describes it in 2005, naming the forward and backward passes and the parallel hardware, with the calm specificity of people who could see what was coming. Ninety-one later inventions cite it.

The convolutional network itself — the architecture being trained — also traces back to the AT&T / Bell Labs work of the previous chapter, where it learned to read cheques. This patent is about making it fast enough to matter, by putting it on the right silicon.

THE IDEA

Suppose your input is an image, or any data laid out on a grid. A plain feed-forward network from Chapter 7 would connect every pixel to every neuron — millions of weights, almost all wasted, because it would have to learn separately that an edge in the top-left corner and an identical edge in the bottom-right corner are the same kind of thing. A convolutional network fixes this with two ideas that are really one idea seen twice.

The first is the filter (or kernel): a small pattern-detector — just a little grid of weights — that you slide across the whole input, computing a score at each position. A filter might learn to fire on vertical edges, another on a particular curve, another on a patch of texture. Because the same filter is slid across every position, the network learns each pattern once and can then detect it anywhere in

the input. This is weight sharing, and it is the source of both the convolutional network's efficiency (far fewer weights) and its power (it generalises across position automatically). The second idea is simply that you stack many filters into a layer, and stack layers on layers, so that early layers detect simple features (edges, blobs) and later layers combine them into complex ones (shapes, objects).

A convolutional network is still trained by the backpropagation of Chapter 7 — forward pass to predict, backward pass to assign blame, gradient step to improve. The only new wrinkle is that the backward pass must respect weight sharing: since one filter touches every position, its gradient is the sum of the blame it accrues across all those positions. The arithmetic is more bookkeeping, but the principle is unchanged.

And here is why graphics hardware matters so much. Sliding a filter across thousands of positions means doing thousands of identical small multiply-and-add operations that do not depend on each other — they can all happen at the same time. A graphics processor is a chip built to do exactly that: thousands of small arithmetic operations in parallel, because that is what rendering a screenful of pixels demands. The match between what convolution needs and what a GPU offers is almost suspiciously perfect. Microsoft's patent saw the match in 2005.

THE STORY

The convolutional network was, for most of its life, an idea ahead of its hardware. It worked — LeNet read cheques in the 1990s — but training a large one was painfully slow on ordinary processors, which do their arithmetic mostly one operation at a time. So the architecture languished in a niche, admired by specialists, while the field's mainstream pursued other methods. The missing ingredient was not a better algorithm. It was speed.

Speed arrived from an unexpected direction. Graphics cards, driven by the enormous consumer demand of the video-game industry, had quietly become some of the most powerful parallel processors on Earth — and they were cheap, because gamers paid for them. Microsoft's 2005 patent is the sound of researchers noticing that this gaming hardware was, by happy accident, a neural-network training engine. The full force of the insight landed in 2012, when a convolutional network trained on GPUs won a major image-recognition contest by a margin so large it stunned the field and triggered the deep-learning boom that is still unfolding. Every large AI system since runs on the descendants of those same parallel chips. The patent that named the technique in 2005 has now expired, a small artifact of a group that saw the future early and wrote it down.

THE CODE

The patentml Conv1D is a one-dimensional convolution — a filter

sliding across a sequence — which is the same idea as the image case with one axis instead of two, and far easier to read. The forward pass is the filter sliding across positions, computing a weighted sum at each:

```
def forward(self, x):
    """x: [length, channels_in] -> output: [out_len, channels_out]"""
    Lout = (len(x) - self.K) // self.stride + 1
    out = []
    for t in range(Lout):                # slide the filter along the input
        t_in = t * self.stride
        a_row = []
        for oc in range(self.Cout):      # for each output filter
            z = self.b[oc]
            for k in range(self.K):       # over the filter window...
                for ic in range(self.Cin):
                    z += self.W[oc][k][ic] * x[t_in + k][ic]
            a_row.append(self.act(z))
        out.append(a_row)
    return out
```

The crucial line is the innermost: the same filter weights `self.W[oc]` are used at every position `t`. That is weight sharing — the filter is learned once and applied everywhere. The backward pass, correspondingly, accumulates each filter's gradient across all the positions it touched:

```
for t in range(Lout):          # every position the filter visited...
    t_in = t * self.stride
    for oc in range(self.Cout):
        delta = d_out[t][oc] * self.d_act(z[t][oc])
        for k in range(self.K):
            for ic in range(self.Cin):
                self.dW[oc][k][ic] += delta * x[t_in + k][ic] # blame
summed over positions
                dx[t_in + k][ic] += delta * self.W[oc][k][ic]
```

In use, a convolutional layer transforms a sequence of feature-vectors into a shorter sequence of higher-level features:

```
from patentml import Conv1D

conv = Conv1D(in_channels=2, out_channels=4, kernel_size=3)
sequence = [[0.1, 0.9], [0.2, 0.8], [0.3, 0.7], [0.4, 0.6], [0.5, 0.5]]
features = conv.forward(sequence)    # 5 positions, 2 channels -> 3
positions, 4 features

print(f"{len(sequence)}×{len(sequence[0])} input -> "
      f"{len(features)}×{len(features[0])} feature map")
```

Four filters, each three steps wide, slide across the five-step input and produce a three-step map of higher-level features — and every one of those filters was applied identically at every position, exactly

the weight-sharing that makes convolution efficient and exactly the parallelism that makes a GPU the right home for it.

THE INHERITANCE

The convolutional network is the tool for data with local structure on a grid — images above all, but also audio, time series, and any signal where nearby values relate and a pattern can appear at any position. Reach for it whenever "the same feature could occur anywhere in the input" is true, because weight sharing turns that property from a curse into an advantage.

The honest note about the public-domain form is the one the module itself makes: for a large convolutional network on real images, you will use a production framework on a real GPU, because that is what the parallel arithmetic demands and what this 2005 patent foresaw. What the dependency-free patentml version gives you is understanding — a convolution you can read line by line, watch slide, and trace the gradient through, with weight sharing visible rather than buried in a compiled kernel. It is the clearest possible way to learn what a convolution actually is before you trust a framework to do a million of them a second. Microsoft wrote down the GPU insight in 2005; the patent has expired; the idea that lit the fuse of the modern era is now a short, readable file.

The next chapter turns from data laid out in space to data laid out in

time — the recurrent networks that process sequences by carrying a memory forward from one step to the next.

CHAPTER 9

MEMORY IN THE WIRES (RECURRENT NETWORKS)

THE FILING

US Patent 6,963,862 — "Method and system for training a recurrent network", governed by the relation $x(k+1) = f(W \cdot x(k))$.

Assignee: Texas A&M University. Filed 2001. Now expired; public domain.

The networks of the last two chapters share a blind spot: they have no memory. A feed-forward or convolutional network maps an input to an output and forgets everything the instant it is done; show it the next input and it starts from scratch, with no sense that anything came before. That is fine for a photograph, where everything you need is present at once. It is hopeless for a sequence — a sentence, a melody, a stream of sensor readings — where the meaning of the current item depends on what preceded it. The Texas A&M patent

addresses exactly this, describing the training of a recurrent network whose defining equation, $x(k+1) = f(W \cdot x(k))$, says something simple and profound: the network's next state is a function of its current state. The output loops back as input. The network remembers.

THE IDEA

A recurrent neural network adds one thing to the feed-forward network: a hidden state that persists from one step of a sequence to the next. At each step, the network looks at two things — the current input and its own hidden state carried over from the previous step — and produces a new hidden state, which it carries forward to the next step. That hidden state is the network's memory: a running summary of everything it has seen so far, updated at every step. Feed it a sentence one word at a time, and by the last word its hidden state has been shaped by the whole sentence.

This is the Elman network, the cleanest form, and the Texas A&M equation captures it exactly: the new state is a learned function of a weight matrix applied to the old state (here, extended to also fold in the new input). Training works by an elegant trick called backpropagation through time: you imagine the recurrent loop "unrolled" into a deep feed-forward network, one copy of the cell per time step, all sharing the same weights — and then you backpropagate through that unrolled network just as in Chapter 7.

The memory makes the network powerful; the unrolling makes it trainable.

Plain recurrent networks have a famous weakness, though: over long sequences, the error signal flowing backward through many steps tends to shrink toward nothing (or, less often, explode), so the network struggles to connect causes to effects that are far apart in time — it forgets the beginning of a long sentence by the time it reaches the end. The fix is gating. A gated recurrent unit, like the GRU implemented here, adds little learned valves that decide, at each step, how much of the past hidden state to keep and how much of the new information to write in. An "update gate" controls how much memory carries over; a "reset gate" controls how much of the old memory influences the new candidate. With gates, the network can learn to hold onto information across long gaps — to remember the subject of a sentence until it finally reaches the verb.

THE STORY

Recurrent networks have a long history, with the Elman and Jordan networks of the late 1980s establishing the basic loop, and the Texas A&M patent sitting in the lineage of efforts to train them reliably. For years they were the natural-but-frustrating choice for sequence data: clearly the right shape for the problem, but hard to train over long ranges because of the vanishing-gradient curse. The gated architectures — the LSTM of the late 1990s and the simpler GRU

that followed — were the breakthroughs that made recurrent networks genuinely work, and for most of the 2010s they were the dominant tool for language, speech, and translation. The voice assistants and machine-translation systems of that era ran largely on gated recurrent networks.

There is a twist to the story worth telling honestly, because it is recent. Around 2017, a non-recurrent architecture called the transformer — which processes a whole sequence at once using "attention" rather than a step-by-step loop — began to displace recurrent networks for the largest language tasks, precisely because it parallelises better on the GPUs of the previous chapter. The systems making headlines today are mostly transformers, not recurrent networks. But the recurrent network is far from obsolete: its step-by-step, constant-memory operation is exactly what you want for streaming data and on-device sequence processing, where you cannot hold the whole sequence at once and must react as data arrives. The patent's loop — next state from current state — remains the natural model of anything that happens over time.

THE CODE

The recurrent cell is the whole idea in one method: combine the current input with the previous hidden state to produce the new hidden state. Here is the patent's Elman cell, which is the Texas A&M equation made literal:

```

def step(self, x, h_prev):
    """One recurrent step: new hidden state from input x and previous
    state h_prev."""
    h = []
    for j in range(self.Dh):
        z = self.b[j]
        for i in range(self.Din):
            z += self.Wx[i][j] * x[i]      # contribution from the current
input
        for k in range(self.Dh):
            z += self.Wh[k][j] * h_prev[k]  # contribution from the
carried memory
        h.append(math.tanh(z))
    return h

```

Those two inner loops are the entire distinction between a recurrent network and a feed-forward one: the second loop, the one over `h_prev`, is the memory. Running it across a sequence simply means threading the hidden state from each step into the next:

```

def forward(self, sequence):
    h = self.cell.zero_state()      # memory starts empty
    outputs = []
    for x_t in sequence:            # one step per item in the sequence
        h = self.cell.step(x_t, h)  # update memory, carry it forward
        y = [sum(self.Wo[j][k] * h[j] for j in range(self.Dh)) + self.bo[k]

```

```
        for k in range(self.Do)]
    outputs.append(y)
    return outputs
```

The gated cell adds the valves. The GRU's update gate z decides how much new information to write versus how much old memory to keep, in a single blending line:

```
return [(1 - z[j]) h[j] + z[j] n[j] for j in range(self.Dh)]
keep old memory    write new candidate
```

When the update gate z is near zero, the unit holds its existing memory almost unchanged — that is how a gated network remembers across long gaps. Using a recurrent network to summarise a sequence into a single decision is then a matter of reading off the last output:

```
from patentml import SimpleRNN

rnn = SimpleRNN(input_size=3, hidden_size=8, output_size=1)
sequence = [[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]]
final = rnn.predict_last(sequence)    # one number, shaped by the
whole sequence

print(f"summary of the sequence: {final[0]:.4f}")
```

The hidden state began empty, was updated by each of the three steps in turn, and the final output reflects all three — the network's memory at work.

THE INHERITANCE

Reach for a recurrent network when your data is a stream that arrives over time and the present depends on the past — sensor readings, control signals, event logs, any sequence you must process as it comes rather than all at once. Its defining virtue, in the streaming setting, is that it uses constant memory: it carries one hidden state forward and never needs to hold the whole history, which is exactly the property you want on a small device watching a never-ending signal. Where the transformers now dominating large-scale language work must see a whole window at once, the recurrent network sips its input one step at a time, forever, in a fixed footprint.

That makes the public-domain, dependency-free form genuinely useful rather than merely educational. A gated recurrent cell in `patentml` is small enough to run on an embedded controller doing real-time sequence work — gesture recognition, anomaly spotting in a sensor stream, simple speech detection — on hardware where a transformer and its framework could never fit. And as ever, the readable version is the best teacher: the memory is right there, in that second loop over the previous state, where you can watch it carry information forward. Texas A&M patented the training of the loop in

2001; it has expired; the network that learned to remember is now a file you can open and a memory you can trace.

The next chapter completes Part II by returning to the engine under all of this. Every network in these chapters learned by stepping its weights down an error gradient — and how you take that step, it turns out, makes an enormous difference. We meet the optimisers: SGD, momentum, RMSProp, and Adam.

CHAPTER 10

THE DESCENT TOOLKIT (SGD, MOMENTUM, RMSPROP, AND ADAM)

THE FILING

US Patent 6,128,606 — "Trainable modular network" with "back-propagated gradients wherein learning rate controls gradient descent updates."

Assignee: AT&T Corporation. Filed 1997. Now expired; public domain.

US Patent 5,835,901 — automatically adjusting the learning rate in real-time neural-network learning.

Assignee: Lockheed Martin Corporation. Filed 1997. 142 forward citations. Now expired; public domain.

Every network in Part II learned the same way: compute the gradient of the error, then step each weight in the direction that reduces it. But

how big a step? And should the step remember the steps before it? These sound like footnotes. They are not. The difference between a naïve step and a clever one can be the difference between a network that trains in minutes and one that never trains at all. The AT&T patent names the essential knob — the learning rate that scales each gradient step — and the Lockheed patent, heavily cited, addresses the next idea: adjusting that learning rate automatically as training proceeds. This chapter is about the family of methods that turn raw gradient descent into something that actually works in practice, and it culminates in the optimiser that trains most of modern machine learning.

THE IDEA

Imagine you are blindfolded on a hillside and want to reach the valley floor. The gradient tells you which way is downhill from where you stand. The simplest strategy — stochastic gradient descent, SGD — is to take a small step in that direction, feel the new slope, step again, and repeat. It works, but it has problems. If your steps are too big you overshoot and bounce around; too small and you crawl. And in a long, narrow valley — common in real error landscapes — pure SGD zig-zags pathetically across the valley walls while making painfully slow progress along its length.

The fixes are a sequence of increasingly clever ideas, each addressing a flaw in the last:

Momentum. Instead of basing each step only on the current slope, accumulate a running "velocity" — keep some of your previous step's direction and add the new gradient to it. Like a ball rolling downhill, you build speed in consistent directions and average out the zig-zags. Momentum alone dramatically smooths and speeds up descent.

RMSProp. Different weights may need different step sizes — some dimensions are steep, others shallow. RMSProp gives each weight its own adaptive learning rate, by tracking a running average of how large that weight's recent gradients have been and dividing its step by the square root of that. Weights with consistently large gradients get smaller, more careful steps; weights with small gradients get larger ones. The step sizes self-tune, per weight.

Adam. Combine the two. Adam — short for "adaptive moment estimation" — keeps both a momentum term (a running average of the gradient, the first moment) and an RMSProp term (a running average of the squared gradient, the second moment), with a small bias correction to keep the early steps honest. It gets momentum's smoothing and RMSProp's per-weight adaptation at once. The result is an optimiser that works well across an enormous range of problems with almost no tuning — which is precisely why it became the default. When a modern network is trained, Adam (or a close relative) is almost always the optimiser turning the gradients into steps.

A final refinement, the subject of Lockheed's patent, is the learning-rate schedule: rather than holding the step size fixed, you change it over training — often warming it up at the start and then annealing it smoothly toward zero at the end, so the network explores boldly early and settles precisely late. It is the same explore-then-refine wisdom that ran through the optimisers of Part I, now applied to gradient descent itself.

THE STORY

These methods accreted over decades, each a response to a felt pain. Momentum dates to Polyak in 1964 — older than most of the field. RMSProp famously entered the world not through a paper but through a slide in a 2012 lecture by Geoffrey Hinton, and spread by word of mouth because it simply worked. Adam was published in 2015 by Kingma and Ba and was adopted with remarkable speed, becoming within a year or two the default optimiser for deep learning, a position it still largely holds. The AT&T and Lockheed patents of 1997 sit earlier in this lineage, claiming the foundational pieces — the learning-rate-scaled gradient step, and the automatic adjustment of that rate during real-time learning.

What is striking, looking back, is how much of the deep-learning revolution rode on these unglamorous improvements. The architectures get the headlines — convolutional networks, recurrent networks, transformers — but none of them would train without

good optimisers underneath. Adam is, in a real sense, as load-bearing as any architecture in the field. It is the quiet machine that turns "here is the gradient" into "here is a better network," billions of times, in nearly every training run on Earth. The pieces it is built from were patented, separately, before the revolution began, and have now expired into the public domain.

THE CODE

The beauty of the patentml optimisers is that each is a small, self-contained step method, and you can read the whole progression of ideas in a few lines each. Here is SGD with momentum — the running velocity is the whole trick:

```
class SGD:
    def step(self, params, grads, param_id=0):
        v = self._v[param_id]          # the running velocity
        new_params = []
        for i, (p, g) in enumerate(zip(params, grads)):
            g_eff = g + self.wd * p      # optional weight decay
            v[i] = self.mom v[i] - self.lr g_eff # keep some
momentum, add the gradient
            new_params.append(p + v[i])
        return new_params
```

RMSProp gives each weight its own adaptive step, dividing by the

root of its recent squared gradients:

```
class RMSProp:
    def step(self, params, grads, param_id=0):
        sq = self._sq[param_id]
        new_params = []
        for i, (p, g) in enumerate(zip(params, grads)):
            sq[i] = self.alpha * sq[i] + (1 - self.alpha) * g * g # track
            # gradient size
            new_params.append(p - self.lr * g / (math.sqrt(sq[i]) +
self.eps)) # adaptive step
        return new_params
```

And Adam combines them — a first moment m (momentum) and a second moment v (RMSProp), each bias-corrected, driving the step together:

```
class Adam:
    def step(self, params, grads, param_id=0):
        ...
        for i, (p, g) in enumerate(zip(params, grads)):
            m[i] = self.b1 * m[i] + (1 - self.b1) * g # 1st moment:
            # momentum
            v[i] = self.b2 * v[i] + (1 - self.b2) * g * g # 2nd moment:
            # RMSProp
            m_hat = m[i] / bc1 # bias correction
```

```

    v_hat = v[i] / bc2
    new_params.append(p - self.lr * m_hat / (math.sqrt(v_hat) +
self.eps))
    return new_params

```

The classic torture test for optimisers is the Rosenbrock function — a long, curved, narrow valley designed to defeat naïve descent. Here all four optimisers race down it from the same start:

```

from patentml import SGD, RMSProp, Adam, AdamW,
numerical_gradient

def rosenbrock(p):
    x, y = p
    return (1 - x) * 2 + 100 (y - x 2) 2    # minimum at (1, 1)

for Opt, name in [(SGD, "SGD"), (RMSProp, "RMSProp"), (Adam,
"Adam"), (AdamW, "AdamW")]:
    p = [-1.0, 1.0]
    opt = Opt(lr=0.01) if Opt is not SGD else Opt(lr=0.001,
momentum=0.9)
    for _ in range(500):
        p = opt.step(p, numerical_gradient(rosenbrock, p))
    print(f"{name:8} -> ({p[0]:.3f}, {p[1]:.3f})  target (1, 1)")

```

Watching the four finish at different distances from the target (1, 1)

after the same five hundred steps is the whole lesson of this chapter made visible: the gradient told each optimiser the same thing at every point, and yet how they turned that gradient into a step decided how close each one got. (Note `numerical_gradient` here — a finite-difference helper that estimates a gradient by nudging each parameter and measuring the change; it is the implementation's built-in way to test an optimiser without hand-deriving calculus.)

THE INHERITANCE

You will use these optimisers, in some form, behind every gradient-trained model in this book and far beyond it — they are not an alternative to the networks of Part II but the engine inside them. The practical wisdom is short and worth carrying: reach for Adam by default, because it works well across a huge range of problems with little tuning, which is exactly why the field adopted it; reach for SGD with momentum when you want maximum control and are willing to tune, since it can sometimes reach a slightly better final solution on large problems; and reach for a learning-rate schedule — warm up, then anneal — whenever a training run is long enough to benefit from exploring boldly early and settling precisely late.

The public-domain, dependency-free form has a particular value here that is easy to miss. Optimisers are usually the most hidden part of a machine-learning system — a single line, `optimizer = Adam(...)`, with all the machinery sealed inside a framework. To see Adam as it

truly is — two running averages and a bias correction, a dozen readable lines — demystifies the single most load-bearing piece of modern training. And because the pieces are small and pure, they will run anywhere a model needs to be trained or fine-tuned in place: on an embedded device adapting to its environment, in a browser, in a locked-down environment with no frameworks allowed. AT&T and Lockheed patented the foundations of adaptive gradient descent in 1997; the patents have expired; the descent toolkit that trains the modern world is now a handful of files you can read in an afternoon.

That completes Part II. We have built the neuron, stacked it into feed-forward, convolutional, and recurrent networks, and met the optimisers that make them all learn. Part III turns to a different and, for many practical problems, more immediately useful family — the classifiers. They include the most-cited patents in this entire book, and they answer the most common question in applied machine learning: given this example, which category does it belong to?

PART III — THE VERDICT: CLASSIFIERS AND THE ART OF DECIDING

Most applied machine learning is, in the end, classification: given an example, which category does it belong to? Is this email spam? Is this transaction fraud? Is this tumour benign? Is this customer about to leave? The eight chapters of this part are the great workhorses that answer such questions — support vector machines, decision trees and forests, boosting, ensembles, online and instance-based learners, and probabilistic classifiers. They include the most-cited patents in this entire book, because these are the techniques that earned their keep in the real world, on real money, for decades. If Parts I and II were about searching and learning in the abstract, Part III is about the deciding that businesses actually pay for. We begin with the method that, for years, was the one to beat: the support vector machine.

CHAPTER 11

THE MARGIN (SUPPORT VECTOR MACHINES)

THE FILING

US Patent 6,327,581 — "Methods and apparatus for building a support vector machine classifier" by solving the quadratic-programming problem through a sweep over training examples.

Assignee: Microsoft Corporation. Filed 1998. 173 forward citations. Now expired; public domain.

The inventor behind this patent is John Platt, and the algorithm it describes — Sequential Minimal Optimization, universally called SMO — is one of those rare pieces of work that quietly became infrastructure. For roughly fifteen years, if you trained a support vector machine in any popular tool — the libsvm library, scikit-learn's classifier, the SVM inside countless commercial

products — you were running Platt's SMO, or a descendant of it. The patent's claim language is unglamorous ("sweeping through training examples... selecting a training example... updating alphas") but what it describes is the trick that made support vector machines practical to train, and the support vector machine was, through the late 1990s and 2000s, frequently the most accurate classifier available. This is a foundational patent, with the citation count to prove it, now expired into the public domain.

THE IDEA

Picture two classes of points scattered on a page — say, dots and crosses — and suppose you can separate them with a straight line. There are infinitely many lines that would do it. Which is best? The support vector machine gives a precise and beautiful answer: the best line is the one that sits as far as possible from both classes — the line with the widest margin, the largest empty corridor on either side before you hit a point. The intuition is that a separator with a wide buffer is more robust; a new point near the boundary is more likely to fall on the correct side. The handful of points that sit right on the edge of the corridor, the ones that "support" the boundary and pin it in place, are the support vectors — and remarkably, they are the only points that matter. You could delete every other training point and get the exact same classifier.

Two ideas extend this from a neat geometric trick into a powerhouse.

The first handles the common case where the classes are not perfectly separable — where a few points stray into enemy territory. Rather than demand a perfect split (impossible, or achievable only by absurd contortions), the SVM allows some violations but charges a penalty for them, controlled by a knob conventionally called C . A large C insists on few violations; a small C tolerates more in exchange for a wider, smoother margin. The second idea is the famous kernel trick, and it is genuinely clever. What if the classes cannot be separated by any straight line — what if the boundary needs to be curved? The kernel trick lets the SVM behave as though it had first projected the data into a much higher-dimensional space, where a straight separator does exist, without ever actually computing that projection. A kernel function — linear, polynomial, or the popular "radial basis function" that measures closeness — quietly stands in for the high-dimensional dot product, and the SVM draws a flat boundary in a space you never have to visit, which corresponds to a curved boundary back in the original space. With an RBF kernel, an SVM can carve out arbitrarily wiggly decision boundaries.

The catch — and the reason Platt's patent mattered so much — is that finding this maximum-margin boundary is a quadratic-programming problem, and solving one directly, for thousands of points, was historically slow and required heavyweight numerical machinery. SMO is the breakthrough. Instead of solving the whole giant problem at once, it repeatedly picks just two of the

unknowns and solves that tiny two-variable subproblem analytically — with a little formula, no heavy solver — then sweeps through the data doing this over and over until everything settles. Many tiny exact steps replace one enormous hard one. That decomposition is the patent, and it is what put SVMs within everyone's reach.

THE STORY

The support vector machine emerged from the work of Vladimir Vapnik and colleagues, rooted in deep statistical-learning theory about why wide margins generalise well — theory that gave the method an unusually rigorous foundation. Through the late 1990s and 2000s, SVMs were often the method to beat: on many real classification problems they matched or exceeded everything else, they came with principled theory, and — thanks to Platt's SMO — they were finally fast to train. This was the era when SVMs and the tree-based methods of the coming chapters dominated applied machine learning, and neural networks were the underdog. For a long stretch, the smart default for a hard classification problem was "try an SVM with an RBF kernel."

The neural-network revolution of the 2010s eventually displaced SVMs from the largest-scale problems — images, language — where deep networks and huge datasets won decisively. But the SVM never stopped being an excellent tool for the enormous middle ground of machine learning: problems with modest amounts of data,

where its strong theoretical guarantees, its robustness, and its tendency not to overfit make it a reliable first choice. Platt's SMO algorithm, the engine that trained essentially all of them for fifteen years, is now public domain — a piece of infrastructure quietly returned to the commons.

THE CODE

The learned form of an SVM is captured in its decision function — a weighted vote among the support vectors, each weighted by its learned coefficient α and its label, with the kernel measuring similarity to the query point:

```
def _decision(self, x):
    """f(x) = sum_i (alpha_i y_i K(x_i, x)) + b"""
    val = self.bias
    for xi, yi, ai in zip(self.X_train, self.y_train, self.alphas):
        if ai > 0:
            # only support vectors contribute
            val += ai yi self.kernel(xi, x)
    return val
```

The training is Platt's SMO, and the structure of the patent's claim is right there in the loop — sweep through the examples, find one that violates the optimality conditions, pair it with another, and update the two alphas analytically:

```
while passes < self.max_passes:
    changed = 0
    for i in range(n):                # "sweeping through training
examples"
        Ei = self._decision(X[i]) - y[i]
        if not (kkt_violated(i)):      # skip examples already
satisfied
            continue
        j = random.choice([k for k in range(n) if k != i]) # pick a
second example
        ...
        eta = 2 * self._K[i][j] - self._K[i][i] - self._K[j][j]
        self.alphas[j] -= y[j] * (Ei - Ej) / eta # analytic two-variable
update
        self.alphas[j] = max(L, min(H, self.alphas[j]))
        self.alphas[i] += y[i] y[j] (aj_old - self.alphas[j])
        ...                          # update the bias from this pair
        changed += 1
    passes = passes + 1 if changed == 0 else 0    # done when a full
sweep changes nothing
```

The kernel trick is the cleanest part of the whole design: it is just a swappable function. Hand the SVM a linear kernel and it draws straight lines; hand it an RBF kernel and it draws curves — same training code, entirely different power. The classic demonstration is XOR, which no straight line can separate but an RBF kernel handles

with ease:

```
from patentml import SVM, rbf_kernel

X = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [-1, 1, 1, -1]                # XOR: no linear separator
exists

svm = SVM(C=1.0, kernel=rbf_kernel(gamma=2.0),
max_passes=50, random_seed=42)
svm.fit(X, y)

print("support vectors:", svm.n_support_vectors())
print("predictions:", svm.predict(X))
correct
```

With a linear kernel this would be hopeless — XOR is the textbook example of a problem no line can solve. Swap in the RBF kernel and the same SMO training finds a curved boundary that separates the classes perfectly, using a handful of support vectors. The kernel did the impossible by pretending to work in a higher-dimensional space the code never actually built.

THE INHERITANCE

The support vector machine is the classifier to reach for on the vast

range of modest-sized problems where you do not have the mountains of data that deep learning demands. With a few hundred or a few thousand examples, an RBF-kernel SVM is often the strongest single model you can train, and it comes with a temperament worth valuing: grounded in solid theory, resistant to overfitting, and dependent only on the support vectors rather than the bulk of the data. When someone hands you a tabular classification problem and asks for a strong baseline fast, "SVM with an RBF kernel" remains, decades on, an excellent answer.

The public-domain, dependency-free form carries a special significance here, because for fifteen years this exact algorithm lived inside the libraries — you called `SVC()` and `SMO` ran where you could not see it. To have Platt's algorithm out in the open, readable, and runnable in pure Python is to hold a piece of machine-learning infrastructure that was, until recently, somebody's property. It will train an SVM in a locked-down environment with no scientific-Python stack, inside a small service, or in a browser — and it is the clearest way to finally see the margin, the support vectors, and the kernel trick that you spent years calling through an API. Microsoft and John Platt made SVMs practical in 1998; the patent has expired; the verdict-machine that ruled applied classification is now yours to read.

The next chapter turns to a completely different style of classifier — one that does not draw a margin in space but asks a series of plain

questions, and whose answers a human can actually read: the decision tree, and the forest grown from many of them.

CHAPTER 12

THE TREE AND THE FOREST (DECISION TREES & RANDOM FORESTS)

THE FILING

US Patent 6,816,847 — "Computerized aesthetic judgment" by training a classifier comprising a decision tree, with feature selection and an ensemble of classifiers.

Assignee: Microsoft Corporation. Filed 1999. 72 forward citations. Now expired; public domain.

The application that produced this patent is charming: Microsoft was trying to teach a computer to judge whether a photograph was aesthetically pleasing. To do it they reached for the decision tree — and, tellingly, for an ensemble of classifiers combined together, which is the seed of the random forest that this chapter builds toward. The decision tree is the most human-legible classifier in this entire book. Unlike the support vector machine's abstract margin or

the neural network's inscrutable weights, a trained decision tree is just a flowchart of yes/no questions that a person can read, follow, and check. That transparency is why trees, and the forests grown from them, remain among the most widely used machine-learning methods in the world — especially on the tabular, spreadsheet-shaped data that makes up the bulk of real business problems.

THE IDEA

A decision tree classifies by asking a sequence of simple questions, each about a single feature. Is the petal length less than 2.5 centimetres? If yes, go left; if no, go right. Is the customer's age over 40? Left or right. Each question splits the data into two groups; each group is split again by a further question; and so on, until you reach a leaf that announces a verdict. To classify a new example, you start at the top and answer the questions until you fall out the bottom into a leaf. The whole model is a flowchart, and you can print it.

The art is in choosing the questions. At each node, the tree considers every feature and every possible threshold and asks: which split makes the two resulting groups as pure as possible — as close as possible to containing only one class each? Purity is measured by one of two classic quantities: entropy (from information theory — how mixed is this group?) or the Gini impurity (the chance of mislabelling a random member). The split that most reduces

impurity — that yields the largest information gain — is chosen, and the process repeats greedily down the tree. A perfectly pure leaf needs no more questions.

Left unchecked, a tree will keep splitting until every leaf is a single training point — at which point it has memorised the training data and will generalise terribly. This is overfitting, and the cures are to limit the tree's depth, to refuse to split groups that are too small, or to prune back branches that do not earn their complexity. (Microsoft's companion patent US6529891, on penalising model complexity, speaks to exactly this discipline of not letting a model grow more elaborate than the data justifies.)

A single tree, however well pruned, has a stubborn weakness: it is unstable. Change a few training points and the greedy choice at the top might flip, producing a completely different tree. The fix is one of the most powerful and beautiful ideas in machine learning: the random forest. Instead of one tree, grow many — hundreds — and have them vote. To make the trees usefully different from one another, you train each on a random bootstrap resample of the data (this is bagging, bootstrap aggregating) and, at each split, let each tree consider only a random subset of the features. Each individual tree is mediocre and overfits in its own idiosyncratic way; but their errors are largely independent, so when they vote, the errors cancel and the truth reinforces. The forest is far more accurate and far more stable than any tree within it. It is the wisdom of crowds, applied to

flowcharts.

THE STORY

The decision tree has deep roots — the ID3 and C4.5 algorithms of Ross Quinlan in the 1980s, and the CART work of Breiman and colleagues around the same time, established the impurity-splitting approach this chapter describes. Trees were popular precisely because managers and domain experts could understand them; a model you can read is a model you can trust and defend, which matters enormously in regulated and high-stakes settings.

The random forest was introduced by Leo Breiman in 2001, building on bagging (also his), and it was something close to a free lunch: take the humble, unstable decision tree, grow a few hundred with a dash of randomness, vote, and obtain one of the most accurate, robust, and low-fuss classifiers ever devised — one that resists overfitting, needs little tuning, handles messy mixed-type data gracefully, and even reports which features mattered. For tabular data, the random forest and the gradient-boosted trees of Chapter 17 have remained, for two decades, the methods to beat. The deep-learning revolution barely touched this territory; on a spreadsheet of numbers and categories, a forest still frequently wins. Microsoft's 1999 patent, reaching for a decision-tree ensemble to judge the beauty of photographs, sits right at the practical heart of this tradition.

THE CODE

Choosing a split means measuring impurity, and `patentml` offers both classic measures plus the information gain that drives the greedy choice:

```
def gini(labels):  
    n = len(labels)  
    counts = Counter(labels)  
    return 1.0 - sum((c / n) ** 2 for c in counts.values())    # impurity  
of a group
```

```
def information_gain(parent, children):  
    parent_h = entropy(parent)  
    n = len(parent)  
    child_h = sum(len(c) / n * entropy(c) for c in children if c)  
    return parent_h - child_h    # how much purer the children are  
than the parent
```

Classifying an example is a literal walk down the flowchart — at each node, compare one feature to one threshold and go left or right:

```
def _predict_one(self, x):  
    node = self.root  
    while not node.is_leaf:  
        if x[node.feature_idx] <= node.threshold:    # answer this node's
```

question

```
        node = node.left
    else:
        node = node.right
    return node.leaf_value                # the verdict at the leaf
```

The forest wraps the tree in bagging and a vote. Each tree is trained on a bootstrap resample, and prediction is a simple majority across the trees:

```
def fit(self, X, y):
    for _ in range(self.n_trees):
        indices = [random.randint(0, len(X) - 1) for _ in range(len(X))]
# bootstrap sample
        X_b = [X[i] for i in indices]
        y_b = [y[i] for i in indices]
        tree = DecisionTree(max_depth=self.max_depth,
n_features=n_feat)
        tree.fit(X_b, y_b)
        self.trees.append(tree)

def predict(self, X):
    all_preds = [tree.predict(X) for tree in self.trees]
    return [Counter(col).most_common(1)[0][0]        # majority vote
across trees
            for col in zip(*all_preds)]
```

In use, the contrast between one tree and a forest is the whole lesson:

```
from patentml import DecisionTree, RandomForest

tree = DecisionTree(max_depth=4)
forest = RandomForest(n_trees=100, max_depth=4, random_seed=1)
tree.fit(X_train, y_train)
forest.fit(X_train, y_train)
```

The single tree is readable but brittle; the forest is more accurate and stable.

```
print("tree :", accuracy(tree.predict(X_test), y_test))
print("forest :", accuracy(forest.predict(X_test), y_test))
```

A bonus the tree gives almost for free is feature importance — by tracking how much impurity each feature's splits removed across the tree (or forest), you get a ranked list of which inputs actually mattered. That single readout, available from the `feature_importances` method, is one of the most practically useful things in applied machine learning: it tells you what your model is paying attention to.

THE INHERITANCE

Reach for a single decision tree when legibility is the priority — when you, or a regulator, or a domain expert, needs to read and audit

the model's logic. A tree is a flowchart; you can print it, walk it, and explain any individual decision exactly. In medicine, credit, and law, that explicability is sometimes worth more than a few points of accuracy.

Reach for a random forest when you want one of the best general-purpose classifiers in existence with minimal fuss — which, for tabular data, is most of the time. It is accurate, robust, hard to overfit, forgiving of messy mixed-type features, and it reports which inputs mattered. For the spreadsheet-shaped problems that dominate real business — churn, fraud, risk, demand — a forest is very often the right first model and the right last one.

The public-domain, dependency-free form makes both genuinely portable. A decision tree is a small flowchart; once trained, evaluating it is a handful of comparisons, fast enough for the tightest embedded budget and simple enough to reimplement on any device. A forest is just many such flowcharts and a vote. Microsoft patented the tree-ensemble classifier in 1999 to judge the beauty of photographs; the patent has expired; the most legible and one of the most useful classifiers in machine learning is now yours to read, grow, and ship.

The next chapter takes the ensemble idea in a sharper direction. Instead of growing many independent trees and voting, what if each new model focused specifically on the mistakes of the ones before it?

That is boosting — and it begins with AdaBoost.

CHAPTER 13

WISDOM OF WEAK CROWDS (ADABOOST)

THE FILING

US Patent 7,421,415 — "Methods and systems for cost-sensitive boosting of weak classifiers" with feature selection, for detection tasks.

Assignee: Siemens. Filed 2005. Now expired; public domain.

The previous chapter combined many classifiers by having them vote independently — each tree grown in ignorance of the others. Boosting is a more cunning idea: build the classifiers in sequence, and have each new one focus specifically on the examples that the previous ones got wrong. It is the difference between assembling a committee of independent experts and training a relay team where each runner is coached on exactly the weaknesses of the last. Siemens' patent applies this to detection — finding objects or events

in noisy data — and pairs it with the valuable side-benefit that boosting naturally performs feature selection, telling you which inputs carry the signal. The algorithm at its core, AdaBoost, was one of the genuine theoretical triumphs of 1990s machine learning, and for years it was the engine behind real-time systems you have certainly used.

THE IDEA

Start with a weak learner — a classifier that barely does better than a coin flip. The classic choice, used here, is a decision stump: a decision tree with exactly one question. A single stump is almost useless on its own. AdaBoost's astonishing claim, proved theoretically, is that you can combine many such near-useless classifiers into one that is arbitrarily strong. Here is how.

Every training example begins with an equal weight. You train your first stump on the weighted data, and it gets some examples right and some wrong. Now comes the key move: you increase the weight of the examples it got wrong and decrease the weight of those it got right. When you train the next stump on this reweighted data, it is effectively told, "never mind the ones we already handle — concentrate on these hard cases." That stump fixes some of the errors, you reweight again to emphasise whatever is still wrong, and train a third stump focused on those. Round after round, the ensemble's attention is dragged toward the stubborn, difficult

examples that earlier rounds could not handle.

Each stump also earns a confidence weight — call it α — based on how accurate it was: a stump that classified the (weighted) data well gets a loud vote in the final ensemble; a barely-better-than-chance stump gets a quiet one. The final classifier is a weighted vote of all the stumps, each speaking with its earned confidence. The mathematics behind the specific reweighting and confidence formulas is elegant — it turns out to be minimising an exponential measure of error, step by step — but the intuition needs no equations: focus successive simple models on the mistakes of their predecessors, then let them vote in proportion to how good they were.

Two happy consequences fall out. First, because each stump splits on a single feature, the running tally of which features the stumps chose, weighted by their confidence, is a natural feature-importance score — this is the feature-selection benefit Siemens' patent highlights. Second, AdaBoost is often surprisingly resistant to overfitting, continuing to improve on fresh data even after it has perfectly classified the training set — a phenomenon that puzzled and delighted researchers for years.

THE STORY

AdaBoost was introduced by Yoav Freund and Robert Schapire in

1995, and it won them the Gödel Prize, one of theoretical computer science's highest honours, because it answered a deep question: could a collection of "weak" learners, each only slightly better than chance, be boosted into a "strong" learner of arbitrary accuracy? The answer — yes, and here is how — was both a beautiful theorem and a wildly practical algorithm.

Its most famous application arrived a few years later. The Viola–Jones face detector of 2001, which made real-time face detection possible on ordinary hardware for the first time, used AdaBoost to select a small number of simple visual features from an enormous pool and combine them into a fast, accurate detector. For most of the following decade, the little box that appeared around faces in your digital camera's viewfinder was AdaBoost at work. Siemens' 2005 patent sits in exactly this lineage — boosting weak classifiers for detection, with the feature-selection that made such systems fast enough to run live. Boosting as a family went on to dominate competitive machine learning through its gradient-boosting descendants, the subject of Chapter 17.

THE CODE

The entire AdaBoost loop is short, and every idea from the explanation is a line or two. Here is the training loop: train a stump on the weighted data, score its confidence, and reweight the examples to emphasise the mistakes:

```

weights = [1.0 / n] * n                # every example starts equal

for t in range(self.T):
    stump = DecisionStump()
    error = stump.fit(X, y, weights)    # best one-question split
on weighted data
    error = max(1e-10, min(1 - 1e-10, error))

    stump.alpha = 0.5 * math.log((1 - error) / error) # confidence: lower
error!' louder vote

    self.feature_importance[stump.feature_idx] += abs(stump.alpha) #
feature selection

    preds = stump.predict(X)
    new_weights = [w * math.exp(-stump.alpha * y_i
the misclassified
                    for w, y_i, pi in zip(weights, y, preds)]
    total = sum(new_weights)
    weights = [w / total for w in new_weights]    # renormalise
    self.stumps.append(stump)

```

Look closely at the reweighting line. The term $y_i \cdot \pi_i$ is positive when the stump was right about example i (label and prediction agree) and negative when it was wrong. So $\exp(-\alpha \cdot y_i \cdot \pi_i)$ shrinks the weight of correctly-classified examples and grows* the weight of

misclassified ones — exactly the "focus on the mistakes" move, in one expression. The final prediction is the confidence-weighted vote of all the stumps:

```
def predict_score(self, X):
    scores = [0.0] * len(X)
    for stump in self.stumps:
        for i, p in enumerate(stump.predict(X)):
            scores[i] += stump.alpha * p      # each stump votes with
its confidence
    return scores
```

Using it, and reading off the features it found important, looks like this:

```
from patentml import AdaBoost

model = AdaBoost(n_estimators=50)
model.fit(X_train, y_train)          # labels are +1 / -1

print("accuracy:", accuracy(model.predict(X_test), y_test))
print("top features:", model.top_features(5))    # which inputs
carried the signal
```

Fifty one-question stumps, each almost useless alone, each trained on the failures of the last, vote together into a classifier that can be

highly accurate — and the `top_features` readout tells you, as a free by-product, which handful of inputs actually mattered. That combination of accuracy and feature insight is precisely what made boosting the engine of real-time detection.

THE INHERITANCE

Reach for AdaBoost when you want a strong classifier built from simple, fast parts, and especially when you also want to know which features matter. Its use of trivially cheap stumps makes the final model fast to evaluate, which is why it powered real-time detection on the modest hardware of two decades ago — and that speed is exactly the property you still want on small or embedded devices today. The built-in feature ranking is a genuine bonus: in one training run you get both a classifier and a shortlist of the inputs that carry your signal, which can guide everything downstream.

There is a deeper inheritance here, too. AdaBoost is the conceptual ancestor of gradient boosting (Chapter 17), the technique that has won more tabular-data competitions than any other and remains a top choice for structured prediction. Understanding AdaBoost's core move — sequentially focus new models on the residual mistakes of the old — is understanding the seed of that whole dominant family. And in its public-domain, dependency-free form, the algorithm is small enough to read in a sitting and to run anywhere: fifty stumps and a weighted vote will fit on hardware where a neural network

never could. Siemens patented cost-sensitive boosting for detection in 2005; Freund and Schapire's underlying algorithm is long public; and the wisdom-of-weak-crowds machine that put the box around your face is now yours.

The next chapter steps back to the broadest form of the combine-many-classifiers idea — and to the single most-cited patent in this entire book.

CHAPTER 14

THE VOTE (VOTING AND STACKING ENSEMBLES)

THE FILING

US Patent 5,613,012 — "Tokenless identification system" using an ensemble of multiple classifiers with voting, for fingerprint classification. 170 claims.

Assignee: SmartTouch, Inc. Filed 1995. 1,182 forward citations.
Now expired; public domain.

This is the most-cited patent in this entire book, and it deserves the spotlight. Over a thousand later inventions point back to it. Filed in 1995 by SmartTouch — a biometric-identification company — its sprawling 170 claims describe a system for identifying people by fingerprint without a token or card, and at its technical heart is an idea this chapter is devoted to: combining the verdicts of multiple different classifiers by voting, to reach a decision more reliable than

any single classifier could give. The companion SmartTouch patent US5764789, itself cited over 800 times, covers weighted combination of pattern matchers. That biometric identification — a high-stakes problem where a wrong answer means letting in an impostor or rejecting a legitimate user — drove such heavily-cited work on classifier ensembles tells you how much the combining of classifiers mattered in practice, long before it became machine-learning orthodoxy.

THE IDEA

The chapters of this part have each built a single kind of classifier — a margin, a tree, a boosted stump. But suppose you have built several different classifiers — an SVM, a random forest, a naive Bayes model — each with its own strengths and its own blind spots. The ensemble idea, in its most general form, is simply: don't pick one; combine them. Where they agree, you can be confident; where they disagree, the combination arbitrates. Because different model families make different kinds of mistakes, their errors tend to be partly independent, and combining them cancels out idiosyncratic errors while reinforcing genuine signal — the same logic that made the random forest work in Chapter 12, now applied across different model types rather than many trees of one type.

There are several ways to combine, in increasing sophistication:

Hard voting is the simplest: each classifier casts one vote for its predicted class, and the majority wins. You can weight the votes — giving your more trustworthy classifiers a louder say — which is exactly the weighted combination the SmartTouch patents describe.

Soft voting is usually better when your classifiers can report not just a verdict but a confidence — a probability for each class. Instead of counting hard votes, you average the probabilities. A classifier that is 95% sure contributes more decisively than one that is barely 51% sure, even if both nominally "voted" the same way. Soft voting uses the full richness of the classifiers' confidence, not just their final answers.

Stacking is the most sophisticated. Rather than combining the base classifiers' outputs with a fixed rule (majority, average), you train a second model — a "meta-learner" — to learn the best way to combine them. The base classifiers' predictions become the input features to this meta-model, which learns, for instance, that the SVM should be trusted on certain kinds of cases and the forest on others. Stacking can squeeze out the last drops of accuracy, at the cost of complexity, and it is a staple of winning competition entries.

The deep principle under all three, and worth holding onto, is diversity. An ensemble only helps if its members are different — if they make different mistakes. Combining ten copies of the same classifier gains you nothing. Combining an SVM, a tree, and a Bayes

model — three genuinely different ways of seeing the data — gains you a great deal. The art of ensembling is the art of assembling disagreement that cancels.

THE STORY

The idea that combining classifiers beats relying on one has deep roots — in the wisdom-of-crowds intuition, in formal results showing that independent weak voters can be aggregated into a strong one (the same theory that underpins boosting), and in the practical experience, accumulated across the 1990s, that ensembles simply won. SmartTouch's heavily-cited patents are an early, commercially-driven instance, applying voting ensembles to the unforgiving problem of biometric identity, where reliability is everything.

The decisive vindication of ensembles came in the era of public machine-learning competitions. When the Netflix Prize — a famous million-dollar contest to improve movie recommendations — was finally won, the victory came from an enormous ensemble blending many models, not from any single brilliant algorithm. The lesson generalised: across competition after competition, the top entries were almost always ensembles, often stacked. Practitioners learned a durable truth — that once you have several decent models, the cheapest remaining accuracy is bought not by inventing a better model but by combining the ones you have. Microsoft's

aesthetic-judgment patent from Chapter 12 made the same move, blending a Bayesian classifier, an SVM, a neural net, and a decision tree. Combining is one of the most reliable tricks in the field, and SmartTouch's thousand-times-cited patent is among its earliest monuments.

THE CODE

The `patentml` voting ensemble shows the two voting styles side by side. Hard voting tallies weighted votes for the predicted classes; soft voting averages the predicted probabilities:

```
def predict(self, X):
    if self.voting == "hard":
        all_preds = [est.predict(X) for est in self.estimators]
        results = []
        for row_preds in zip(*all_preds):
            votes = {}
            for pred, w in zip(row_preds, self.weights):
                votes[pred] = votes.get(pred, 0) + w    # weighted
majority vote
            results.append(max(votes, key=votes.get))
        return results
    else:
        # soft voting
        all_probab = [est.predict_proba(X) for est in self.estimators]
        results = []
```

```
    for i in range(len(X)):
        class_probs = {}
        for probas, w in zip(all_probas, self.weights):
            for cls, prob in probas[i].items():
                class_probs[cls] = class_probs.get(cls, 0) + w * prob #
average confidence
        results.append(max(class_probs, key=class_probs.get))
    return results
```

The elegance is that the ensemble does not care what its members are — any object with a fit and predict method qualifies. That lets you combine completely different classifiers from across this book into a single voter:

```
from patentml import VotingEnsemble, SVM, RandomForest,
rbf_kernel

ensemble = VotingEnsemble(
    estimators=[
        SVM(kernel=rbf_kernel(gamma=1.0)),
        RandomForest(n_trees=50),
        RandomForest(n_trees=50, max_depth=3), # a deliberately
different forest
    ],
    voting="hard",
    weights=[1.0, 1.5, 1.0], # trust the deep forest a little
```

```
more
)
ensemble.fit(X_train, y_train)
print("ensemble accuracy:", accuracy(ensemble.predict(X_test),
y_test))
```

Three different models — a kernel SVM and two forests of different depth — each see the data their own way and make their own mistakes; the weighted vote arbitrates, and the combination is typically steadier and more accurate than any member alone. The `StackingEnsemble` in the same module goes further, feeding the base models' predictions into a trained meta-learner that discovers the best way to blend them.

THE INHERITANCE

Reach for an ensemble whenever you have already built a few decent models and want more accuracy cheaply. This is one of the most dependable moves in all of applied machine learning: you rarely lose by combining good, diverse models, and you frequently gain. When the stakes are high — biometric identity, fraud, medical decisions, anything where a single model's blind spot could be costly — the robustness of a vote across diverse classifiers is worth a great deal, which is precisely why SmartTouch built its identification system this way.

The practical wisdom is to chase diversity: combine models that are genuinely different in kind, so their errors are genuinely different, because that is what makes the combination greater than its parts. Soft voting when your models report confidence, stacking when you want to wring out the last accuracy and can afford the complexity.

The public-domain, dependency-free form is especially natural for ensembling, because the ensemble is just a thin layer of voting logic over whatever classifiers you already have — and in `patentml`, every classifier you might want to combine is right there, in the same dependency-free library, sharing the same simple interface. You can assemble an SVM, a forest, a boosted model, and a Bayes classifier into one voter, all in pure Python, all runnable in a locked-down environment or a browser. SmartTouch patented the voting ensemble for fingerprint identity in 1995 and earned over a thousand citations for it; the patent has expired; the combine-the-crowd machine at the heart of so many winning systems is now yours to assemble.

The next chapter turns to a classifier built for a world that never stops sending data — one that learns continuously, updates itself from its own confident predictions, and gave the world one of its first and most beloved applications of machine learning: the spam filter.

CHAPTER 15

LEARNING ON THE FLY (THE ONLINE CLASSIFIER)

THE FILING

US Patent 6,161,130 — "Technique which utilizes a probabilistic classifier to detect junk e-mail by automatically updating a training set and re-training the classifier"

Assignee: Microsoft Corporation. Filed 1998. 896 forward citations. Now expired; public domain.

Here is a patent you have benefited from thousands of times without knowing it. Its citation count — nearly nine hundred — marks it as one of the genuinely influential documents in this book, and its subject is one of the first applications of machine learning that ordinary people relied on every day: the spam filter. The patent describes a probabilistic classifier that does something most of the classifiers in this part do not — it keeps learning after deployment.

The claim is precise: "automatically updating the training set to include classification of message content for an incoming message; re-training the classifier based on the updated training set." In plain terms: the filter watches new email, and when it is confident about a message, it quietly adds that message to its own training data and updates itself. It improves itself, continuously, from the very stream it is processing. That is online learning, and the spam filter is its most famous success.

THE IDEA

Every classifier so far in this book has been batch-trained: you gather all your labelled data, train the model once, and then deploy a frozen thing that never changes. That works when the world is stable. But the world often is not — and spam is the perfect example, because spammers deliberately change their tactics to evade filters. A spam filter trained once and frozen would be obsolete within weeks. What you need is a classifier that adapts as it goes.

The classifier underneath is naive Bayes, one of the oldest and most charming methods in machine learning, and worth understanding in its own right. It rests on a delightfully simple application of Bayes' rule and one "naive" assumption: that the features are independent given the class. For text, the features are words. The model learns, from labelled examples, how often each word appears in spam versus legitimate mail ("ham"). To classify a new message, it asks: given

the words in this message, and given how spammy or hammy each of those words tends to be, what is the probability this message is spam? The word "viagra" pushes hard toward spam; "meeting" pushes toward ham; the message's overall score is the combination. The independence assumption — pretending each word's evidence is separate from the others' — is technically false (words are correlated), but it works astonishingly well in practice, and it makes the model fast, tiny, and trivially updatable. Probabilities are multiplied as sums of logarithms to avoid numerical underflow, and a smoothing constant prevents a single never-before-seen word from zeroing out the whole estimate.

The Microsoft patent's contribution is the online, self-training loop wrapped around this classifier. When a new message arrives, the filter predicts its class and its confidence. If the confidence clears a high threshold, the filter treats its own prediction as a new labelled example and folds it into its counts — learning from itself. Because naive Bayes is just a tally of word counts per class, "retraining" is as cheap as incrementing some counters; there is no expensive refit. The filter therefore tracks the drifting language of spam in real time, learning the new tricks as they arrive, at almost no computational cost. (A note of caution the design implies: self-training on your own confident predictions risks compounding a mistake, so the confidence threshold must be high and, in practice, real systems also fold in genuine user corrections — the "this is spam" button.)

THE STORY

The naive Bayes spam filter is a landmark in the popular history of machine learning. In the late 1990s and early 2000s, as email became universal and spam became a plague, the probabilistic filter — trained on your own mail, learning which words signalled junk — was the technology that turned the tide. It was famously effective, easy to implement, and gave millions of people their first direct, daily benefit from a learning algorithm. Paul Graham's widely-read essay "A Plan for Spam" popularised the approach and set off a wave of adoption. Microsoft's patent, filed in 1998 and cited nearly nine hundred times, is a foundational piece of that story, capturing the crucial self-updating mechanism that kept the filters ahead of the spammers.

The broader idea — online learning, a model that updates continuously from a stream rather than training once on a fixed batch — has only grown in importance. Recommendation systems that adapt to your latest clicks, fraud detectors that adjust to new attack patterns, any system facing a world that shifts under its feet: all need to learn on the fly. Naive Bayes remains a favourite for this because its update is so cheap and its behaviour so transparent. Microsoft's spam-filter patent is, in a sense, the public-domain template for an entire style of adaptive machine learning.

THE CODE

The naive Bayes scoring is the heart of it — combine the prior probability of each class with the evidence of each word, all in log-space for numerical safety:

```
def predict_proba(self, text):
    tokens = self._tokenise(text)
    log_probs = {}
    for cls in self.class_counts:
        lp = math.log((self.class_counts[cls] + self.smoothing) /
prior: how common is this class?
                        (self.total_docs + self.smoothing * len(classes)))
        cls_total = sum(self.feature_counts[cls].values()) +
self.smoothing * vocab_size
        for tok in tokens:
            count = self.feature_counts[cls].get(tok, 0) + self.smoothing
            lp += math.log(count / cls_total)    # evidence: how
spammy/hammy is this word?
        log_probs[cls] = lp
    # convert log-scores back to probabilities (softmax)
    max_lp = max(log_probs.values())
    probs = {cls: math.exp(lp - max_lp) for cls, lp in
log_probs.items()}
    total = sum(probs.values())
    return {cls: p / total for cls, p in probs.items()}
```

The online self-training loop — the Microsoft patent's actual claim

— is strikingly short. Predict, and if confident, learn from your own prediction:

```
def predict_and_autotrain(self, text):
    """US6161130: predict, then auto-add to the training set if
    confident."""
    label, conf = self.predict(text)
    auto = False
    if self.auto_retrain and conf >= self.threshold:  # only if we're
sure
        self.train(text, label)                      # fold this example into the
model
        self.auto_labelled += 1
        auto = True
    return label, conf, auto
```

And train itself is just incrementing counters — which is why "retraining" costs almost nothing:

```
def train(self, text, label):
    self.class_counts[label] += 1
    self.total_docs += 1
    for tok in set(self._tokenise(text)):
        self.feature_counts[label][tok] += 1
```

Seeded with a handful of examples, the filter classifies new mail and

grows as it goes:

```
from patentml import OnlineBayesClassifier

clf = OnlineBayesClassifier(confidence_threshold=0.75)
clf.train_batch([
    ("buy cheap pills online now", "spam"),
    ("meeting tomorrow at 3pm", "ham"),
    ("win a million dollars today", "spam"),
    ("lunch plans for friday?", "ham"),
])

label, conf, auto = clf.predict_and_autotrain("free offer click here")
print(f"[{label}] confidence={conf:.2f}
learned-from-itself={auto}") # !' spam
```

With only four seed examples, the filter already leans correctly on a new message — and, if confident enough, files that message into its own memory, a fraction better prepared for the next one. Multiply that across a real inbox's stream and you have a classifier that quietly teaches itself the evolving shape of spam.

THE INHERITANCE

Reach for an online classifier whenever your data arrives as a stream and the world keeps changing — spam, fraud, content moderation,

anomaly detection in live systems, anything where a model frozen at training time would slowly rot. The defining virtue is adaptation at near-zero cost: because naive Bayes updates by incrementing counters, it can learn from every new example in real time without ever stopping to retrain, which is exactly what a non-stationary world demands.

Naive Bayes also remains a superb baseline and a superb teacher in its own right, online or not. It is fast, it is tiny, its probabilities are interpretable, it needs very little data to start being useful, and it is often shockingly competitive on text. When you face a new text-classification problem, naive Bayes is the right thing to try first — it sets the bar everything else must clear.

The public-domain, dependency-free form is a natural fit for this kind of work, because the whole model is a dictionary of word counts and a scoring loop — small enough to run inside an email client, a tight embedded service, a browser extension, or any place that must classify a stream with no heavyweight machine-learning stack and no round-trip to a server. Microsoft patented the self-updating spam filter in 1998 and earned nearly nine hundred citations for it; the patent has expired; one of the first learning algorithms ordinary people ever relied on is now yours to read and to run.

The next chapter turns to a classifier with no training phase at all —

one that makes its decisions by simply looking at which known examples a new point most resembles.

CHAPTER 16

THE NEIGHBOURHOOD (K-NEAREST NEIGHBOURS)

THE FILING

US Patent 6,374,251 — "Scalable system for clustering of large databases" via efficient nearest-neighbour search over partitioned data.

Assignee: Microsoft Corporation. Filed 1998. 187 forward citations. Now expired; public domain.

Most classifiers in this book do real work before they ever see a test example: the SVM solves a quadratic program, the tree greedily splits, the network descends a gradient for thousands of steps. The k-nearest-neighbours classifier does something almost cheeky by comparison — it does no training at all. It simply remembers every example it was given, and when you ask it to classify a new point, it looks up which stored examples that point most resembles and lets

them vote. It is the laziest classifier imaginable, and it works remarkably well. The catch — and the subject of Microsoft's heavily-cited patent — is that "looking up which examples a point most resembles" becomes painfully slow when you have remembered millions of them. The patent is about making that nearest-neighbour search scalable through clever partitioning of the data, which is exactly the problem that turns a trivially simple idea into an engineering challenge worth 187 citations.

THE IDEA

The k-nearest-neighbours method rests on an assumption so simple it barely seems like an algorithm: things that are close together tend to be alike. To classify a new point, find the k stored examples nearest to it — by straight-line distance, usually — and let those k neighbours vote. If you ask for the 5 nearest neighbours and 4 of them are labelled "cat," you call the new point a cat. That is the whole method. There is no model, no parameters, no training loop; the "training" is just storing the data, which is why the method is called lazy — it defers all the work to query time.

A few choices shape its behaviour. The number k is the main dial: a small k (say 1) makes the classifier sensitive to every local quirk and noisy point; a large k smooths the decision by consulting a wider neighbourhood, at the risk of blurring genuine fine structure. The distance metric defines what "near" means — usually ordinary

Euclidean distance, but the choice matters, and it makes the method acutely sensitive to the scale of your features: if one feature ranges over thousands and another over fractions, the large-ranged feature will dominate the distance and drown the other out, so features generally must be normalised first. The same method works for regression, too: instead of voting on a class, the neighbours average their numeric values, optionally weighted so that closer neighbours count for more.

The deep appeal of k-nearest-neighbours is that its decision boundary can be arbitrarily complicated without any extra effort — it is implicitly defined by the data itself, curving and folding however the examples demand, with no functional form to choose. The deep problem is cost at query time. A naive lookup compares the query point against every stored example, which is fine for thousands of points but ruinous for millions. This is where Microsoft's patent earns its citations: by partitioning the stored data into a spatial structure — the implementation here includes a ball tree, which groups points into nested spheres — you can rule out huge swathes of the data without examining them, and find the true nearest neighbours while looking at only a small fraction of the points. The idea is simple; making it fast is the invention.

THE STORY

Nearest-neighbour classification is one of the oldest ideas in pattern

recognition, dating to the 1950s and 60s, and it has never gone away precisely because it is so simple, so general, and so often good enough. It is the natural baseline against which fancier methods must justify themselves: if your elaborate model cannot beat "just look at the nearest few examples," you should wonder whether the elaboration was worth it.

The enduring research story, the one Microsoft's patent belongs to, is the fight against query-time cost. As databases grew into the millions and billions of records, the naive scan became impossible, and a whole literature grew up around approximate and partitioned nearest-neighbour search — ball trees, k-d trees, locality-sensitive hashing, and modern vector-search systems. That last phrase deserves emphasis, because the idea has come roaring back: the vector databases at the heart of today's AI systems — the ones that let a large language model retrieve relevant documents by embedding them as points and finding the nearest ones — are, at bottom, industrial-strength nearest-neighbour search. The lazy classifier of the 1950s, scaled up by exactly the kind of partitioning Microsoft patented in 1998, is now infrastructure for the most advanced AI applications in the world. Few ideas in this book have aged so well.

THE CODE

The classifier itself is almost comically short, because there is no

training to speak of — fit just stores the data:

```
class KNNClassifier:
```

```
    def fit(self, X, y):
```

```
        self._X = [row[:] for row in X]  # "training" is simply
```

```
remembering everything
```

```
        self._y = list(y)
```

```
        return self
```

```
    def _neighbours(self, x):
```

```
        dists = [(self._dist(x, self._X[i]), i) for i in range(len(self._X))]
```

```
        dists.sort()                # nearest first
```

```
        return dists[: self.k]      # keep the k closest
```

```
    def predict(self, X):
```

```
        results = []
```

```
        for x in X:
```

```
            nbrs = self._neighbours(x)
```

```
            labels = [self._y[i] for _, i in nbrs]
```

```
            results.append(Counter(labels).most_common(1)[0][0])  #
```

```
majority vote of neighbours
```

```
        return results
```

That is the brute-force version — exact, simple, and fine for modest data. It can even report class probabilities, as the fraction of neighbours in each class, which makes it a well-behaved member of

the soft-voting ensembles of Chapter 14. For large data, the same library provides the BallTree and FastKNN that implement the partitioned search Microsoft's patent describes, finding the same neighbours far faster by skipping points that cannot possibly be close.

Using it is as direct as the idea:

```
from patentml import KNNClassifier

knn = KNNClassifier(k=3)
knn.fit(X_train, y_train)          # instant — just stores the data
print("accuracy:", knn.score(X_test, y_test))
```

No epochs, no convergence, no tuning beyond the choice of k — fit is instantaneous, and the classifier's intelligence lives entirely in the data it remembers and the distances it computes at query time.

THE INHERITANCE

Reach for k -nearest-neighbours first as a baseline — the honest, near-zero-effort yardstick that every more complex model must beat to justify itself. Reach for it, too, when your decision boundary is genuinely intricate and you would rather let the data define it than commit to a functional form, and when training cost must be zero because new labelled examples arrive constantly and you cannot

afford to retrain — adding data to a k-NN model is just appending to a list.

The honest caveats travel with it: it is hungry at query time and at memory (it stores everything), it demands normalised features, and it degrades in very high dimensions, where "nearest" loses meaning as all points drift to similar distances. Those are real limits, and the partitioned-search inheritance — ball trees and their descendants — is precisely the answer to the first of them.

The public-domain, dependency-free form is especially fitting here because the method is its data structure. A k-NN classifier in `patentml` is a stored list and a distance function; the ball tree is a few hundred lines of partitioning. You can run exact nearest-neighbour search on a small device, embed a tiny example-based classifier in an application with no machine-learning stack, or use the readable ball tree to finally understand the spatial-partitioning trick that powers the vector databases now central to modern AI. Microsoft patented scalable nearest-neighbour search in 1998 and earned 187 citations as the idea quietly became infrastructure; the patent has expired; the oldest and laziest classifier in the book — and the engine of today's vector search — is yours.

The next chapter returns to boosting, the sequential-ensemble idea from Chapter 13, and follows it to its most powerful modern form — the gradient-boosted trees that have won more competitions on

tabular data than any other method.

CHAPTER 17

BOOSTING THE GRADIENT (GRADIENT BOOSTING MACHINES)

THE FILING

US Patent 7,031,530 — "Compound classifier for pattern recognition applications", building a sequential ensemble in which each stage corrects the residual errors of the previous stages using a differentiable loss.

Assignee: Lockheed Martin Corporation. Filed 2001. 188 forward citations. Now expired; public domain.

In Chapter 13, AdaBoost built a strong classifier by training simple models in sequence, each focused on the mistakes of the last.

Gradient boosting is the grown-up, generalised form of that idea, and it is, for tabular data, arguably the most successful machine-learning technique ever devised. Lockheed's patent describes its essence precisely: a "compound classifier" built in stages, "where each stage

corrects residuals of the previous stage using a differentiable loss function." That single sentence contains the whole leap from AdaBoost to gradient boosting — the move from reweighting examples to explicitly fitting the residual errors, and from a specific loss to any differentiable loss. The methods descended from this idea — XGBoost, LightGBM, and their kin — have won more data-science competitions on structured data than everything else combined, and they remain, in 2026, the first thing a serious practitioner reaches for when the data lives in a spreadsheet.

THE IDEA

Gradient boosting builds an ensemble of small regression trees, one at a time, where each new tree is trained to fix what the ensemble-so-far still gets wrong. Here is the idea built up from the simplest case, regression.

Start with the dumbest possible model: just predict the average of all the target values for every input. This is wrong for almost every example, and the amount by which it is wrong — the gap between the true value and the current prediction — is called the residual. Now train a small tree not to predict the target, but to predict those residuals — to learn the pattern in the current model's errors. Add that tree's predictions (scaled down by a small learning rate, for caution) to the running model. The model is now a little less wrong. Compute the new, smaller residuals, and train another tree to predict

those. Add it in. Repeat, dozens or hundreds of times. Each tree nudges the ensemble's prediction a little closer to the truth by attacking the errors that remain. The final prediction is the initial average plus the small, accumulated contributions of every tree.

Why is it called gradient boosting? Because fitting the residuals turns out to be exactly equivalent to taking a step of gradient descent — the same idea from Chapter 10 — but in the space of functions rather than the space of weights. The residuals, for squared-error loss, are the negative gradient of the loss. This is the deep insight Lockheed's patent names with "differentiable loss function": once you see boosting as gradient descent on a loss, you can swap in any differentiable loss and boost toward it. For classification, you use a logistic loss, and the "residuals" become the difference between the true label and the model's current predicted probability — but the loop is identical: fit a tree to the current errors, add it in, repeat.

Three safeguards make it work in practice and appear throughout the implementation. The learning rate scales down each tree's contribution, so the ensemble approaches the truth in many small, cautious steps rather than a few reckless ones — slower, but far less prone to overshooting into overfitting. Shallow trees (often just a few levels) keep each individual learner weak, which is the whole point of boosting — strength comes from the sequence, not the members. And subsampling — training each tree on a random fraction of the data — injects helpful randomness that improves generalisation, an

idea borrowed from the bagging of Chapter 12.

THE STORY

The intellectual lineage runs from AdaBoost (1995) through a crucial reframing by Jerome Friedman around 1999–2001, who recognised boosting as gradient descent in function space and thereby generalised it to arbitrary differentiable losses — giving birth to the gradient boosting machine. Lockheed's 2001 patent sits right at this moment of generalisation, describing the staged, residual-correcting, differentiable-loss compound classifier that Friedman's insight made possible.

Then came the competition era, and gradient boosting dominated it. When the data-science community converged on online competition platforms in the 2010s, a particular high-performance implementation of gradient-boosted trees — XGBoost — became so consistently victorious on tabular problems that it was almost a cliché: the winning solution, again and again, was XGBoost (or a close relative like LightGBM), often inside an ensemble of the kind from Chapter 14. The reasons are exactly the strengths described above: gradient-boosted trees handle messy, mixed-type, real-world tabular data superbly, capture complex interactions automatically, require only moderate tuning, and squeeze remarkable accuracy from structured data. The deep-learning revolution, for all its triumphs in images and language, never displaced gradient boosting from its

tabular throne. On a spreadsheet of features, in 2026, it is still very often the best thing you can do — and its foundational patent has now expired.

THE CODE

The regression case is the clearest, and the `patentml` training loop is the idea made literal: compute residuals, fit a tree to them, add the tree's scaled prediction to the running model, repeat.

```
def fit(self, X, y):
    self._F0 = sum(y) / len(y)          # start with the mean of the
    targets
    F = [self._F0] * len(y)             # current prediction for every
    example
    self._trees = []

    for _ in range(self.n_estimators):
        residuals = [y[i] - F[i] for i in range(len(y))] # what the model
        still gets wrong
        tree = _build_tree(Xs, residuals_subsample, self.max_depth,
        self.min_samples)
        self._trees.append(tree)
        for i in range(len(y)):
            F[i] += self.learning_rate * _predict_tree(tree, X[i]) # nudge
        toward the truth
```

```
return self
```

The residual line — $y[i] - F[i]$ — is the entire conceptual core: each tree learns the current errors, not the original target. Prediction simply replays the accumulation: the starting mean, plus every tree's scaled contribution.

```
def predict(self, X):
    return [self._F0 + self.learning_rate * sum(_predict_tree(t, x) for t
in self._trees)
           for x in X]
```

For classification, the only change is that the "residuals" become the gap between the label and the model's predicted probability (via a sigmoid), implementing the logistic loss the Lockheed patent's "differentiable loss" language anticipates. Using either is the familiar fit-and-predict:

```
from patentml import GBMRegressor, GBMClassifier

reg = GBMRegressor(n_estimators=100, learning_rate=0.1,
max_depth=3)
reg.fit(X_train, y_train)
print("R^2:", reg.score(X_test, y_test))    # variance explained

clf = GBMClassifier(n_estimators=100, learning_rate=0.1,
```

```
max_depth=3)
clf.fit(X_train, y_train)
print("accuracy:", accuracy(clf.predict(X_test), y_test))
```

A hundred shallow trees, each trained on the errors left by the previous ninety-nine, accumulate into a model that is typically far more accurate than any single tree or even the random forest of Chapter 12 — the difference between a committee voting independently (the forest) and a relay team each fixing the last runner's mistakes (boosting).

THE INHERITANCE

For tabular data — the spreadsheet-shaped problems that make up most of real-world machine learning — gradient boosting is very often the right answer, full stop. Churn prediction, credit risk, fraud, demand forecasting, click-through rates, insurance pricing, medical risk scores: across this enormous and economically central category, gradient-boosted trees are the reigning champion, and have been for a decade. If someone hands you a table of features and a column to predict, gradient boosting should be at the very top of your list of things to try — frequently it is also the last thing you need to try.

The honest practical notes: it has more knobs than a random forest (the number of trees, the learning rate, and tree depth interact, and you will want to tune them), and the cautious-small-steps philosophy

means it trains more slowly than a forest. But the accuracy is usually worth it, and the method rewards the tuning.

The public-domain, dependency-free form is a genuine gift, because gradient boosting is the method people most often assume requires a heavyweight specialist library. To have a readable gradient-boosting machine in pure Python — the residual loop right there, the differentiable-loss idea visible in the difference between the regressor and the classifier — is both the best possible way to understand the technique that wins the competitions, and a way to run it where the big libraries cannot go: locked-down environments, tiny services, embedded scoring, the browser. Lockheed patented the staged, residual-correcting compound classifier in 2001 and earned 188 citations as it became the tabular-data champion; the patent has expired; the single most reliable tool for structured prediction is now yours to read and run.

The final chapter of this part turns from squeezing out maximum accuracy to reasoning transparently under uncertainty — the Bayesian network, a classifier you can not only use but interrogate.

CHAPTER 18

THE PROBABILISTIC DETECTIVE (BAYESIAN NETWORKS & NAIVE BAYES)

THE FILING

US Patent 6,490,698 — "Multi-level decision-analytic approach to failure and repair in hierarchical systems", building a probabilistic graphical model and propagating belief through it for structured inference.

Assignee: Microsoft Corporation. Filed 1999. 102 forward citations. Now expired; public domain.

The classifiers of this part have, for the most part, been black boxes that output a verdict. The Bayesian network is something different in kind: a model you can interrogate. It represents the world as a graph of causes and effects, with the strengths of those relationships written down as probabilities, and it lets you reason in any direction — from causes to likely effects, or, more powerfully, from observed

effects back to their probable causes. Microsoft's patent applies exactly this to diagnosis: given a hierarchical system that has failed, and some observed symptoms, reason backward to the most probable underlying fault and the best repair. That is the Bayesian network's signature move — probabilistic detection — and it is why this family powers medical diagnosis, fault-finding, risk analysis, and any setting where you must reason about hidden causes from partial evidence, and explain your reasoning afterward.

THE IDEA

A Bayesian network is a graph. Each node is a variable — Rain, Sprinkler, Wet Grass, Disease, Symptom — and an arrow from one node to another means the first directly influences the second. Crucially, the graph encodes which variables are directly related and, by their absence, which are only related through others. Each node carries a small table — a conditional probability table — that says how likely each of its states is, given the states of its parents. Rain might be "yes" 30% of the time. Wet Grass, given rain, might be "wet" 90% of the time; given no rain, only 20%. Lay out a handful of such nodes and tables and you have compactly specified a full joint probability distribution over the whole system — one that would take an enormous table to write out directly, but that the graph factorises into small, intelligible, local pieces.

The magic is inference: asking questions of the network in the

presence of evidence. You observe that the grass is wet. What is the probability it rained? This is reasoning backward, from effect to cause, and it is exactly what a diagnostician does — observe symptoms, infer the disease. The network computes it by considering all the ways the evidence could have come about, weighted by their probabilities, and reading off the answer. You can pour in whatever you happen to know — some variables observed, others hidden — and the network will give you the updated probability of anything you care about, automatically accounting for every path of influence between them. Observing one thing can change your belief about another in subtle, multi-step ways that the graph tracks correctly. This is "belief propagation," the patent's phrase, and it is reasoning under uncertainty done properly.

This chapter's classifier cousin, naive Bayes (the engine of the spam filter in Chapter 15), is in fact the simplest possible Bayesian network: one cause node (the class) pointing at many effect nodes (the features), with the "naive" assumption that the features are independent given the class. A full Bayesian network drops that assumption and lets you model the real web of dependencies among variables — at the cost of more modelling effort, but with far richer reasoning. The deep contrast with everything else in this part is that a Bayesian network is built from knowledge as much as from data: you can encode what an expert knows about how a system works directly into the graph and the tables, and the network will reason with it correctly. It is the bridge between human knowledge and

probabilistic machinery.

THE STORY

Bayesian networks were developed largely through the work of Judea Pearl in the 1980s, for which he later received the Turing Award — the same honour given to the deep-learning pioneers of Chapter 7, which tells you the field regards Pearl's contribution as comparably foundational. Pearl gave a rigorous, computable framework for reasoning about uncertainty and, eventually, about causality itself — the difference between mere correlation and genuine cause and effect, which is one of the deepest problems in all of science and which his graphical models made tractable.

Through the 1990s and 2000s, Bayesian networks became the tool of choice for expert systems that reason under uncertainty: medical diagnosis systems that weigh symptoms against diseases, fault-diagnosis systems for machinery and software (Microsoft's patent, and indeed the famous printer-troubleshooter built into Windows of that era, are squarely in this tradition), risk models in finance and engineering, and decision-support systems of all kinds. Their great virtue, in an age increasingly worried about inscrutable AI, is transparency: a Bayesian network's reasoning can be inspected, explained, and audited, because it is made of explicit, human-readable probabilities and causal structure rather than millions of opaque weights. As the field grapples with the demand

for explainable AI, the Bayesian network — a model that reasons in a way a human can follow and justify — has a renewed claim on our attention. Microsoft's diagnostic patent has now expired into the public domain.

THE CODE

Building a network is a matter of declaring nodes, their parents, and their probability tables — the model reads almost like a description of the world. Here is the classic rain-and-wet-grass example:

```
from patentml import BayesianNetwork

bn = BayesianNetwork()
bn.add_node("Rain", states=["yes", "no"])
bn.add_node("Wet", states=["yes", "no"], parents=["Rain"])    #
rain influences wet grass

bn.add_cpt("Rain", {(): {"yes": 0.3, "no": 0.7}})            # it rains
30% of the time
bn.add_cpt("Wet", {("yes",): {"yes": 0.9, "no": 0.1},          # given
rain: usually wet
                  ("no",): {"yes": 0.2, "no": 0.8}})          # given no rain:
usually dry
```

The reasoning happens in query, which asks for the probability of a

target variable given whatever evidence you have observed — and the engine works by enumerating over the hidden variables, weighting each possible world by its joint probability:

```
def query(self, target, evidence=None):
    """Compute P(target | evidence) by summing over the hidden
    variables."""
    target_states = self._nodes[target]
    hidden = [n for n in order if n != target and n not in evidence]
    dist = {}
    for ts in target_states:
        current_evidence = {**evidence, target: ts}
        dist[ts] = self._enumerate(hidden, current_evidence, order) #
    sum over hidden worlds
    norm = sum(dist.values()) + 1e-12
    return {k: v / norm for k, v in dist.items()} #
    normalise to probabilities
```

The detective's question — reasoning backward from effect to cause — is now a one-liner:

We observe wet grass. How likely is it that it rained?

```
posterior = bn.query("Rain", evidence={"Wet": "yes"})
```

```
print(posterior) # {'yes': ~0.66, 'no': ~0.34}
```

Before observing anything, the chance of rain was 30%. After

observing wet grass, the network revises that belief upward to about 66% — automatically, correctly, by weighing how readily rain versus no-rain could have produced the wet grass it saw. That upward revision is probabilistic inference: evidence flowing backward along the arrow of causation to update belief in the cause. Scale this from two nodes to a medical network of diseases and symptoms, and the same machinery becomes a diagnostician.

THE INHERITANCE

Reach for a Bayesian network when you must reason about hidden causes from partial, uncertain evidence — and explain your reasoning. Diagnosis is the archetype: symptoms observed, disease inferred, with every step of the inference open to inspection. Reach for it, too, when you have genuine expert knowledge to encode — a doctor's understanding of how diseases produce symptoms, an engineer's model of how components fail — because the Bayesian network is almost unique among the methods in this book in letting you build a model from knowledge as much as from data, and in reasoning correctly with a mix of both.

Its defining inheritance, in an era anxious about opaque AI, is transparency. A Bayesian network does not merely output a probability; it outputs a probability you can trace back through explicit causes and explicit numbers, audit, and defend to a regulator or a patient. As explainability becomes a requirement rather than a

luxury, that property is worth more than ever.

The public-domain, dependency-free form suits this perfectly, because a Bayesian network is a small, human-readable structure — nodes, arrows, and probability tables — and the inference is a transparent sum over possibilities, all of it plain Python you can read and trust. You can embed a diagnostic network in a device, a decision-support tool, or an application with no machine-learning stack, and you can show anyone exactly how it reasons. Microsoft patented the structured-diagnosis Bayesian network in 1999; Pearl's framework underneath it is a Turing-Award-winning foundation of the field; the patent has expired; and the probabilistic detective is now yours to build and to interrogate.

That completes Part III, and with it the three great movements of Volume One — the evolutionary optimisers, the neural networks, and the classifiers. One chapter remains: a capstone that assembles these public-domain parts into a single working system, and then opens the door to Volume Two, where the comfortable assumption behind everything we have built — that someone already knew the right answer — falls away.

PART IV — A WORKING SYSTEM

. . .

CHAPTER 19

A PREDICTOR FROM PUBLIC-DOMAIN PARTS

We have spent eighteen chapters meeting algorithms one at a time. This final chapter of Volume One does something different: it assembles them. Real machine learning is never a single algorithm called once; it is a pipeline — a sequence of steps that takes raw data in one end and produces trustworthy decisions out the other. Every piece of that pipeline is in patentml, drawn from the expired patents of the preceding chapters, and in this chapter we build the whole thing from those parts and nothing else. Then, having built it, we will find the assumption it secretly rests on — and that assumption is the doorway to Volume Two.

THE ANATOMY OF A REAL PIPELINE

A working supervised-learning system has four stages, and we have met the parts for all of them.

First, prepare the features. Raw data is rarely fit to learn from directly. Features arrive on wildly different scales — an age in years beside an income in pounds — and, as Chapter 16 warned, a method that measures distances will let the large-numbered feature drown the rest. So you normalise: rescale every feature to a comparable range. You may also have more features than you need, some of them noise; you can select the informative ones. The `feature_engineering` module — itself drawn from Microsoft's dimensionality-reduction patents — gives you scalers and selectors for exactly this.

Second, choose and train a model. This is the heart of Parts I–III. For tabular data, the wise default is the gradient-boosting machine of Chapter 17 or the random forest of Chapter 12; for a strong, theory-backed classifier on modest data, the support vector machine of Chapter 11; for something legible, a single decision tree; for a streaming problem, the online classifier of Chapter 15. You will often train several.

Third, combine and evaluate honestly. Having trained several models, the ensemble of Chapter 14 lets you combine them into something better than any one. And — this is the step beginners skip and professionals never do — you must evaluate on data the model has never seen, held back from training, because a model's accuracy on the data it was trained on is a flattering lie. The gap between training accuracy and held-out accuracy is the measure of overfitting, the recurring villain of this book.

Fourth, deploy and, where the world drifts, keep learning — the online, self-updating discipline of Chapter 15.

BUILDING IT

Here is a complete supervised pipeline, assembled entirely from public-domain parts. We normalise the features, train three different models, combine them by voting, and evaluate on a held-out set:

```
from patentml import (StandardScaler, RandomForest,
                      GBMClassifier,
                      SVM, rbf_kernel, VotingEnsemble)
```

1. Prepare features — put everything on a comparable scale

```
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)      # same scaling, learned
from train only
```

2 & 3. Train several different models and combine their votes

```
ensemble = VotingEnsemble(
    estimators=[
        RandomForest(n_trees=100, max_depth=6),      # Chapter 12
        GBMClassifier(n_estimators=100, max_depth=3), # Chapter
17
        SVM(kernel=rbf_kernel(gamma=0.5)),          # Chapter 11
```

```
],  
    voting="hard",  
)  
ensemble.fit(X_train_s, y_train)
```

```
4. Evaluate honestly — on data the models never saw  
preds = ensemble.predict(X_test_s)  
print("held-out accuracy:", accuracy(preds, y_test))
```

Read what just happened. Three genuinely different ways of seeing the data — a committee of trees voting independently, a relay of trees each fixing the last's mistakes, and a margin drawn in an implicit high-dimensional space — were each trained on identically-scaled features, and their verdicts were pooled by a weighted vote. The scaler was fit on the training data alone and merely applied to the test data, so no information leaked backward from the test set. And the number printed at the end is the only one that matters: performance on examples the system has never encountered. This is not a toy. It is the skeleton of a real production classifier, and every bone of it came from a patent that has expired into the public domain. No frameworks. No dependencies. No permission required.

THE BRIDGE

Step back and notice the assumption humming under everything in

Volume One. Every algorithm we built — the genetic algorithm climbing toward a known fitness, the neural network descending toward known targets, every classifier in Part III trained on examples someone had already labelled — every one of them assumed that the right answer was available. Somebody knew which emails were spam. Somebody knew which tumours were malignant. Somebody knew the value of x that the formula should reproduce. The machine's job was only to generalise from answers it was given. We have spent a whole volume in the comfortable world of the answer key.

The world is not so generous. Most of the data in existence is unlabelled. Nobody has sorted your photos by the faces in them; nobody has grouped your customers into segments; nobody has told the spacecraft what its true position was at each instant; nobody has marked which transactions, among millions, are the anomalies. And in the richest problems of all, there is no fixed answer to learn at all — only consequences that arrive after you act, rewards and penalties from which you must infer what good behaviour even is. When no one will hand you the answer, every method in this volume falls silent. You cannot minimise an error you cannot measure. You cannot fit a label you do not have.

That silence is where Volume Two begins. *Patterns and Decisions* is about everything that happens once the answer key is taken away: finding structure in unlabelled data by clustering and density;

tracking hidden state through time when the truth is never observed directly; reasoning about uncertainty itself; learning to act from reward rather than instruction; and pulling meaning out of raw language with no one to define it. The patents are just as old, just as expired, and just as freely yours. But the questions change entirely — from learn the answer I am showing you to find the answer no one knows.

Turn to Volume Two when you are ready to work without the answer key.

APPENDIX A: THE PATENTS OF VOLUME ONE

Every algorithm in this volume traces to a United States patent that has expired into the public domain. Each can be read in full, for free, on Google Patents by searching its number. Citation counts are forward citations — the number of later patents that cite this one as prior art, a rough measure of influence.

Chapter	Patent	Assignee	Filed	Cites	Subject
---------	--------	----------	-------	-------	---------

--- --- --- --- --- ---

1	US5970487	Mitsubishi Electric	1997	—	Genetic algorithm machine (hardware)
---	-----------	---------------------	------	---	--------------------------------------

1	US6912587	AT&T	2001	—	Constraint-weighted GA fitness
---	-----------	------	------	---	--------------------------------

2	US6477444	Fuji Xerox	2000	70	Genetic programming, subtree crossover
---	-----------	------------	------	----	--

3	US6212427	Kennedy & Eberhart	1999	—	Particle swarm
---	-----------	--------------------	------	---	----------------

optimisation |
| 3 | US6484115 | Storn & Price | 1999 | — | Differential evolution |
| 4 | US7219040 | General Electric | 2002 | 51 | Constraint-based
optimisation |
| 4 | US7447669 | Nanyang Polytechnic | 2004 | — | Ant colony
optimisation |
| 5 | US5835901 | Lockheed Martin | 1997 | 142 | Neuroevolution |
| 5 | US7162461 | SAS Institute | 2005 | — | Hybrid NN, activation
search |
| 6 | US7047169 | University of Illinois | 2002 | 52 | Estimation of
distribution (EDA) |
| 7 | US6128606 | AT&T | 1997 | 53 | Modular network, backprop |
| 8 | US7747070 | Microsoft | 2005 | 91 | CNNs trained on GPUs |
| 9 | US6963862 | Texas A&M | 2001 | — | Recurrent network
training |
| 10 | US6128606 / US5835901 | AT&T / Lockheed | 1997 | 53 / 142 |
Learning-rate-scaled gradient descent |
11	US6327581	Microsoft (Platt)	1998	173	SVM via SMO
12	US6816847	Microsoft	1999	72	Decision tree + ensemble
13	US7421415	Siemens	2005	—	Boosting weak classifiers
14	US5613012	SmartTouch	1995	1,182	Voting ensemble
(most cited)					
15	US6161130	Microsoft	1998	896	Online self-training
classifier					
16	US6374251	Microsoft	1998	187	Scalable
nearest-neighbour search |

| 17 | US7031530 | Lockheed Martin | 2001 | 188 | Gradient boosting
machine |
| 18 | US6490698 | Microsoft | 1999 | 102 | Bayesian network
diagnosis |

APPENDIX B: READING A PATENT

A patent has two parts that matter. The specification is the long descriptive body — background, drawings, and a detailed account of how the invention works; this is where the real teaching lives, and it is meant to be complete enough that a skilled person could build the thing. The claims, at the end, are the legal heart — numbered sentences that define precisely what the patent protected. Claim 1 is usually the broadest. When this book quotes a patent's claim, that is what it is quoting.

To check that a patent has expired, find its filing date (not the grant date) and add twenty years, the term for a US utility patent of this era. Every patent in this volume was filed before 2006, so all have expired. You can confirm any of this yourself: search the patent number on Google Patents, where the filing date, the claims, the full specification, and the forward-citation list are all freely available.

APPENDIX C: RUNNING THE CODE

Install the companion library once — `pip install patentml` — and every listing in this volume is importable and runnable, with no other dependency. Each algorithm lives in a single, self-contained file you can read end to end; to find the one behind a chapter, look for the module named in that chapter's code listings (for example from `patentml import RandomForest`). Because the whole library is pure standard-library Python, the same files run unmodified on a server, a Raspberry Pi, or — through Pyodide — in a web browser, and can be trimmed to run on a microcontroller. To see any algorithm demonstrated, run its module directly; each ends with a small worked example.

CHAPTER 20

INDEX OF ALGORITHMS · INDEX OF COMPANIES AND INVENTORS

(Compiled at typesetting.)

End of Volume One — Optimisers and Predictors.

The story continues in Volume Two — Patterns and Decisions.