Machine Learning & Deep Learning

David Picard

david.picard@enpc.fr

École Nationale des Ponts et Chaussées

Fall 2026

Contents

List of Definitions

List of Theorems

List of Algorithms

List of Listings

List of Exercises

This is the textbook accompanying the course ***Machine Learning & Deep Learning*** at Ecole Nationale des Ponts et Chaussées. It contains mostly extended version of the material presented during the class, as well as few exercises.

Machine Learning is a wide domain that extends from very theoretical abstractions to very fine-tuned practical implementation. In my experience, it is difficult to cover both because in addition to the required skills, it appeals mostly to different people. I have tried a middle-ground here, a bit of theory, a bit of practical implementations, and thus a minimal knowledge of maths and programming is required. I have marked 🦉 the section that are more difficult than average regarding the maths level of the textbook, and 🐍 the ones that are more difficult regarding code. I am sure some people will prefer ones over the other.

LLMs have been used for many aspects of this book. They did the latex template and the graphic style of the plots since both styling and matplotlib are foreign languages to me, and I find the default style ugly. They also wrote most of the code that is shown in the listing and the code that was run to produce the results presented here. Often, I found that the code that I write is a bit too obscure because of years of bad habits inherited from `C/C++` clunky optimization, like reusing variables or inlining small computations. The LLMs are much more legible and consistent in notations than me, but I did modify almost all files nonetheless, mostly because of small taste-related or cosmetic things. LLMs were also used to chase typos, and believe me, there were a lot of them.

1

Introduction to Machine Learning

This first chapter introduces the main concepts of ***Machine Learning*** and motivates the need for such a new and radically different science[1]. It provides most of the formal definition that will be required throughout the textbook as well as a very light introduction to the main theories. It also provides a concrete ***ML*** algorithm example in the form of the ***Nearest Neighbor*** rule.

1.1 What is Machine Learning?

The world is absurdly complex. So much so that most of the interesting phenomena are very difficult to understand, even more difficult to forecast, and the scientific method has been developed to improve the first in hope that it trickles into improvements on the second. In fact, it could be argued that science is not so much about prediction as it is about understanding. The great mathematician Henri Poincaré wrote:

> *Le savant n'étudie pas la nature parce que cela est utile ; il l'étudie parce qu'il y prend plaisir et il y prend plaisir parce qu'elle est belle.*
>
> — Henri Poincaré, *Science et Méthode* (1908)

In this quote, Poincaré motivates the job of a scientist by uncovering the beauty – in the sense of mathematical aesthetics – of the world. There is obviously a lot to say about this realistic interpretation of physics, but the main point is that utility is not the goal, it is merely a byproduct of the exercise. Since then, the beauty of nature as revealed itself to be much more complex than initially thought and one look at the standard model or at Einstein's field equations is enough to figure out that most

[1] One could even question whether ***Machine Learning*** is a science!

humans will never be able to enjoy their elevated aesthetics. Moreover, these complex models are very far from explaining very complex phenomena like those that could be of interest to many people, like sport bets for example. Or the economic impacts of Climate Change. Or the best candidate molecules against a new virus.

When the system is too complex, we all seem to be ready to exchange some of that beauty for a bit of precision. John von Neumann, arguably one of the most influential scientists of the 20th century, wrote the following:

> *It is that the sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work – that is, correctly to describe phenomena from a reasonably wide area. Furthermore, it must satisfy certain esthetic criteria – that is, in relation to how much it describes, it must be rather simple.*
>
> — John von Neumann, *Method in the Physical Sciences* (1955)

Although von Neumann keeps the aesthetics argument of Poincaré, the focus is definitely on the accuracy of the resulting model. Of course, accuracy is not equal to usefulness and von Neumann argues in another essay[2] that utility is quasi-independent from this endeavor, ***i.e.***, many results were obtained because they were intriguing and beautiful, but many others were obtained because they were the solution to some practical problem.

But, what does this historical perspective on science have to do with ***Machine Learning***, you may ask? Surely this textbook is not about epistemology, although it never hurts. You are right, and we are briefly presenting these historical views to give context to our proposed informal definition: ***Machine Learning is the art of automating the creation of models from data such that they make accurate predictions***. Notice how this definition is a big leap of faith compared to the past: the very concept of understanding or explaining is absent, and so is the beauty of the resulting model. All that matters is the accuracy, regardless of plausibility. In short, ***ML*** is not about correctly describing the world, it is about making accurate predictions. Second, notice that the source for building the model is not human intuition but solely data. Of course data was used in the past – for example, Kepler used the Rudolphine tables from data collected by Brahe – but human intuition was the key ingredient. Not so much in ***ML***, as we hope to automate the discovery of the correct model from the data alone. We will come back to this regularly, as the trade-off between data and human intuition is very prominent in the field.

With that in mind, we will now conclude this short introduction by defining a formal setting and introducing various notations.

1.1.1 Formal setting

Let us consider a pair of random variables (x, y) related through a joint, but unknown, probability $P(x, y)$. We would like to find a model f, such that when applied to x, we obtain y with a reasonable probability. In other words, we want an f that predicts y from x with a low probability of error. In order to do so, we first have to define that error:

[2] *The Role of Mathematics in the Sciences and in Society*, 1954.

Definition 1.1 (01-loss)
Given a prediction $f(x)$ and a target y, the 01-*loss* of f at x is

$$\ell(f(x), y) = 0 \text{ iff } f(x) = y, \text{ else } 1.$$

Definition 1.2 (True Risk)
Let (x, y) be a pair of random variables with joint probability density $P(x, y)$, the ***true risk*** of a predictor f is

$$\mathcal{R}_f = \mathbb{E}\left[\ell(f(x), y)\right] = \int \ell(f(x), y) dP(x, y),$$

with ℓ the 01-loss.

In the present chapter, we only consider the 01-loss, but most results generalize to other losses, notably most of the ones we will study in subsequent chapters.

The crux of the machine learning problem, is that the true risk is not computable since we do not know $P(x, y)$. If we were to know it, life would be much simpler as we would just have to scan all possible y for a given x and choose f as the predictor that maximizes $P(x, f(x))$ for all x. Such error corresponds to the irreducible error that one makes when y is not deterministically tied to x and is called the ***Bayes error***:

Definition 1.3 (Bayes error)
Let (x, y) be a pair of random variables with joint probability density $P(x, y)$, the irreducible error that a predictor f can make is

$$\mathcal{R}_{\min} = \mathbb{E}\left[\ell(\arg\max_z P(x, z), y)\right] = \int \ell(\arg\max_z P(x, z), y) dP(x, y),$$

and is called the ***Bayes error***.

In particular, if the process linking y to x is deterministic, then the irreducible error vanishes:

Proposition 1.4 (Minimal error of deterministic processes)
$\mathcal{R}_{\min} = 0$ *if and only if the process* $x \mapsto y$ *is deterministic, that is* $\forall z \neq y, P(x, z) = 0$.

Proof. Split the expectation into the cases where $\arg\max_z P(x, z) = y$ and $\arg\max_z P(x, z) \neq y$. The first case always sums to 0 by definition of the 01-loss. In the second case, since $z \neq y$ then the 01-loss is always equal to 1 and the integral, containing only non-negative values, is 0 if and only if $\forall z \neq y, P(x, z) = 0$, or equivalently $x \mapsto y$ is deterministic. ■

These definitions are useful for setting up and studying the learning problem, but they unfortunately do not provide practical use. For that, we have to turn to existing data.

1.2 Empirical Risk Minimization

Instead of relying on an abstract distribution $P(x, y)$, let us consider a set of n samples $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$ that we obtained through an unspecified data collection process, but that we assume is independently and identically distributed from $P(x, y)$.

Using $\mathcal{A}$, we now would like to find a predictor f such that $f(x)$ is often y in the sense of the 01-loss. More practically, let us consider $\{f_\theta\}_{\theta \in \Theta}$ a predictor family of parameter space Θ and determine

whether any f_θ is a good predictor on $\mathcal{A}$. We can do so by computing the error on $\mathcal{A}$, which we call the ***Empirical Risk***:

DEFINITION 1.5 (Empirical Risk)
Let $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$ be a dataset of pairs (x_i, y_i) sampled according to $P(x, y)$, the ***empirical risk*** of a predictor f_θ is

$$\hat{\mathcal{R}}_{\mathcal{A}}(f_\theta) = \frac{1}{n} \sum_{(x,y) \in \mathcal{A}} \ell(f_\theta(x), y)$$

with ℓ the 01-loss.

In particular, since the samples in $\mathcal{A}$ are i.i.d., we have the following proposition:

PROPOSITION 1.6 (Unbiased and consistent estimator of the true risk)
The empirical risk is an unbiased and consistent estimator of the true risk, that is

$$\mathbb{E}[\hat{\mathcal{R}}_{\mathcal{A}}(f\theta)] = \mathcal{R}_f,$$
$$\lim_{n \to \infty} \hat{\mathcal{R}}_{\mathcal{A}}(f_\theta) - \mathcal{R}_f = 0.$$

This leads us to the first principle of machine learning, which consists in choosing f_θ in $\{f_\theta\}_{\theta \in \Theta}$ such that it minimizes $\hat{\mathcal{R}}_{\mathcal{A}}(f_\theta)$. This is called ***Empirical Risk Minimization***, or ***ERM*** for short.

DEFINITION 1.7 (Empirical Risk Minimization)
Given a family of predictors $\{f_\theta\}_{\theta \in \Theta}$ of parameter space Θ and a dataset $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$ of pairs (x_i, y_i), the Empirical Risk Minimization principle consists in returning the predictor $f_{\theta^\star}$ such that

$$\theta^\star = \arg\min_{\theta \in \Theta} \hat{\mathcal{R}}_{\mathcal{A}}(f_\theta)$$

Notice how the ERM makes no assumption on $P(x, y)$ other than 1) it exists, and 2) we can sample $\mathcal{A}$ from it. And yet, the ERM already guarantees in the limit that it will correspond to the true risk of the considered predictor. In essence, the ERM is a purely data-driven process: Given some amount of data, we search a parameter space for the best possible fit.

One may argue that the nature of the parameter space is important, as it constrains the search and thus limits the solutions attainable by ERM. This observation leads to another definition of importance. Indeed, if one chooses to restrict the search space based on some intuition, or ***a priori*** knowledge, on the problem, then the process is no longer purely data-driven. It has what is called an ***inductive bias***:

DEFINITION 1.8 (Inductive bias)
The Inductive bias is the set of assumptions, prior knowledge, or preferences that a machine learning algorithm uses in addition to data.

In the strict sense, there can never be a machine learning algorithm without ***any*** inductive bias, since the mere fact of selecting x and y is already making some assumption on the problem, but in practice, we usually use the term to denote much stronger constraints on the problem.

1.3 Generalization

It is of no surprise that the ERM only provides a good predictor on $\mathcal{A}$, as, after all, we already know the y on $\mathcal{A}$. What we truly are interested in, is the accuracy of our predictor on pairs $(x, y) \notin \mathcal{A}$. If our predictor selected using the ERM is accurate outside of $\mathcal{A}$, we say that it *generalizes*.

To quantify how much a predictor generalizes, we use the difference between the true risk and the empirical risk, a quantity known as the *Generalization Gap*:

Definition 1.9 (Generalization Gap)
Let $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$ be a dataset of pairs (x_i, y_i) sampled according to $P(x, y)$, the *generalization gap* of a predictor f is

$$\Gamma_f = \mathcal{R}_f - \hat{\mathcal{R}}_{\mathcal{A}}(f).$$

Since the true risk is not computable, the generalization gap is an abstract quantity that cannot be used in practice. It is, however, a precious tool at the theoretical level since it is the quantity that we would like to guarantee for any ERM algorithm.

1.3.1 Generalization Bounds

Providing guarantees can be done by proving upper bounds to the generalization gap. Such bounds depend on the size of $\mathcal{A}$ and (optionally) on the characteristic of the ERM algorithm that is used. This activity has occupied theoretical machine learning researchers for most of the late 20th century and the early 21st one.

To show an example of how such bounds can be obtained, we focus on a simpler case where a perfect predictor exists and the goal is to find it. Such an easy case is formalized with the following assumption:

Definition 1.10 (Realizability assumption)
A predictor family $\mathcal{H}$ is said to satisfy the *realizability assumption* for a given data distribution $\mathcal{D}$ if there exists $h^\star \in \mathcal{H}$ such that $\mathcal{R}_{h^\star} = 0$.

Notice that under the realizability assumption, the following proposition holds:

Proposition 1.11 (Null empirical risk under realizability assumption)
If the realizability assumption holds for $\mathcal{H}$ under the distribution $\mathcal{D}$, then there exists $h^\star \in \mathcal{H}$ such that $\hat{\mathcal{R}}_{\mathcal{A}}(h^\star) = 0$ almost surely, for any $\mathcal{A} = \{(x, y)\}$ sampled from $\mathcal{D}$.

Proof. Since $\mathcal{R}_{h^\star} = 0$ and is an integral of non-negative values only, then $\ell(h^\star(x), y) = 0$ for all $(x, y) \sim \mathcal{D}$ almost surely. Then the 01-loss also sums to 0 almost surely on any finite union of such samples. ■

Using the realizability assumption, we can prove a very simple generalization bound for the ERM, that takes into account only the size of the predictor family and the size of the training set $\mathcal{A}$:

Theorem 1.12 (PAC generalization bound)
Let $\mathcal{H}$ be a finite predictor family, $\delta > 0$, $\varepsilon > 0$ and m such that $m \geq \frac{\ln(|\mathcal{H}|/\delta)}{\varepsilon}$, then for any distribution $\mathcal{D}$ such that the realizability assumption holds, following the ERM on a dataset

$\mathcal{A} = \{(x_i, y_i) \sim \mathcal{D}\}_{i \leq m}$ *yields a predictor* $h_{\mathcal{A}}$ *such that*

$$P\left[\Gamma_{h_{\mathcal{A}}} > \varepsilon\right] \leq \delta.$$

Proof. First, notice that because of Proposition 1.11, $\Gamma_{h_{\mathcal{A}}} > \varepsilon$ is equivalent to $\mathcal{R}_{h_{\mathcal{A}}} > \varepsilon$, since there exists a classifier achieving 0 empirical risk and $h_{\mathcal{A}}$ is a minimizer of the empirical risk. Thus, when following the ERM, the probability of having a generalization gap greater than ε is the same as the probability of sampling a ***misleading*** training set $\mathcal{A}$ that leads to a ***bad*** predictor with a true risk greater than ε. Let us define the set of bad predictors as

$$\mathcal{H}_B = \{h \in \mathcal{H} \mid \mathcal{R}_h > \varepsilon\},$$

and the set of misleading training sets as

$$\mathcal{M} = \{\mathcal{A} \mid \exists h \in \mathcal{H}_B, \hat{\mathcal{R}}_{\mathcal{A}}(h) = 0\}.$$

In particular, misleading training sets can be obtained by scanning all the bad predictors:

$$\mathcal{M} = \cup_{h \in \mathcal{H}_B} \{\mathcal{A} \mid \hat{\mathcal{R}}_{\mathcal{A}}(h) = 0\}.$$

Let us denote $h_{\mathcal{A}}$ the predictor obtained by following the ERM on $\mathcal{A}$ and notice that a dataset that has the ERM leading to a bad classifier is necessarily a misleading dataset because of the realizability assumption, that is

$$\{\mathcal{A} \mid \mathcal{R}_{h_{\mathcal{A}}} > \varepsilon\} \subseteq \mathcal{M}.$$

The probability of sampling such dataset is bounded from above using the previous union definition:

$$P(\mathcal{A} \mid \mathcal{R}_{h_{\mathcal{A}}} > \varepsilon) \leq \sum_{h \in \mathcal{H}_B} P(\mathcal{A} \mid \hat{\mathcal{R}}_{\mathcal{A}}(h) = 0).$$

Because the samples in $\mathcal{A}$ are i.i.d., the right-hand term corresponds to an (un)lucky streak of good predictions. And since h is a bad classifier, the probability of producing a good prediction is less or equal to $1 - \varepsilon$. Combining these observations leads to the following upper-bound on the probability of observing a misleading training set:

$$P(\mathcal{A} \mid \mathcal{R}_{h_{\mathcal{A}}} > \varepsilon) \leq \sum_{h \in \mathcal{H}_B} (1 - \varepsilon)^m.$$

Extending the number of terms in the sum to $|\mathcal{H}|$, recalling that $(1 - \varepsilon)^m \leq e^{-\varepsilon m}$ and using the definition of m completes the proof. ■

Such bounds are originating from the ***Probably Approximately Correct*** (PAC) theory due to Valiant (1984) and are among the first to quantify generalization, and thus learning, theoretically under minimal assumption on the data distribution of the learning algorithms. The number of samples m is called the ***sample complexity*** and different learning algorithms can be more or less ***sample efficient***, that is, requiring more or less samples to achieve good error rates.

This one in particular tends to be impractical. For example, if we consider a predictor on ***rgb*** images of size 64×64 that consists of a template image, and taking into account the binary encoding of such images, we get $|\mathcal{H}| = 2^{64 \times 64 \times 24}$ and m in the order of 6.8 million samples for a 1% chance of doing 1% error. This is already a lot for such small images and such a simple predictor, and it gets worse for more complex problems.

1.3.2 *Cross-validation & Model Selection*

Instead of relying on theoretical bounds to quantify generalization, empirical clues can be computed to obtain much more practical tools. The general idea is very simple: By sampling a second dataset i.i.d., we can compute a second empirical risk and see whether the first one was just a lucky shot or if both results are consistent. Because the datasets are i.i.d., the probability of independently sampling two misleading datasets is the product of the probability of sampling a misleading dataset, which hopefully gives us much more confidence in the results.

In fact, the problem is much deeper than just putting another coin in the dataset slot machine. Because we use the ERM on $\mathcal{A}$, we favor predictors that do well on $\mathcal{A}$ and this tells us little on the performances that we are interested in, that is, how it performs outside of $\mathcal{A}$. Let us imagine a predictor that returns y for all $x \in \mathcal{A}$ and 0 otherwise. Such predictor has $\hat{\mathcal{R}}_{\mathcal{A}} = 0$ and yet its predictions are terrible for most samples not in $\mathcal{A}$ (assuming y is not always 0 in that case). How do we prevent the ERM principle from selecting such a bad predictor? The short answer is that we cannot, we have to find another way.

This highlights the two faces of the problem: on the one hand we can use the ERM to search a family of predictors for a good one, on the other hand the ERM is not effective at telling us how the selected predictor generalizes. Since that second property is the one we are truly interested in, we have to perform what is called ***model selection*** based on their true performances.

One way of estimating the true performances is to perform ***cross-validation***, that is, splitting the set $\mathcal{A}$ into a training set and a validation set. The training set is used to perform the ERM algorithm, and the validation set is used to estimate the true performances.

We detail two cross-validation algorithms below:

Algorithm 1.1 K-Fold cross-validation

Input: Training set $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$, ERM algorithm, number of folds K
Output: Average error of h

$n \leftarrow |\mathcal{A}|$
for $k \leftarrow 0$ **to** $K - 1$ **do**
  $\mathcal{A}_k \leftarrow \{(x_i, y_i) \in \mathcal{A} \mid i \notin [kn/K, (k+1)n/K]\}$
  $\mathcal{V}_k \leftarrow \{(x_i, y_i) \in \mathcal{A} \mid i \in [kn/K, (k+1)n/K]\}$
  $h \leftarrow \mathrm{ERM}(\mathcal{A}_k)$
  $e_k \leftarrow \mathrm{error}(h, \mathcal{V}_k)$
return $\sum e_k / K$.

Algorithm 1.2 Random split cross-validation

Input: Training set $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$, ERM algorithm, number of splits K, training ratio r
Output: Average error of h

$n \leftarrow |\mathcal{A}|$

for $k \leftarrow 0$ **to** $K-1$ **do**
  $\pi \leftarrow \text{randperm}(n)$
  $\mathcal{A}_k \leftarrow \{(x_{\pi(i)}, y_{\pi(i)}) \in \mathcal{A} \mid i \leq rn\}$
  $\mathcal{V}_k \leftarrow \{(x,y) \in \mathcal{A} \mid (x,y) \notin \mathcal{A}_k\}$
  $h \leftarrow \text{ERM}(\mathcal{A}_k)$
  $e_k \leftarrow \text{error}(h, \mathcal{V}_k)$
return $\sum e_k / K$.

The main difference between the two is that K-Fold outputs an error that has used every sample only once, whereas Random Split can re-use some samples. However, Random Split gives you control over the training set size, if that is a parameter of importance.

The extreme case of K-Fold is when $K = |\mathcal{A}|$, which is appropriately called *leave-one-out* (LOO) validation. LOO is very costly, but if the computation budget allows it, it is very informative. In fact, Bousquet and Elisseeff (2002) proved the following generalization bound for LOO in a landmark paper:

> THEOREM 1.13 (LOO generalization bound)
> *Let $\mathcal{A}$ be a training set of size n and $h_{\mathcal{A}}$ be the predictor obtained by an ERM algorithm using a loss bounded in $[0, M]$, then with probability $1-\delta$ over the choice of $\mathcal{A}$*
>
> $$\mathcal{R}_{h_{\mathcal{A}}} \leq \mathcal{R}_{loo}(h_{\mathcal{A}}) + \gamma + (4n\gamma + M)\sqrt{\frac{\ln 1/\delta}{2n}},$$
>
> *with $\mathcal{R}_{loo}$ the leave-one-out error and provided*
>
> $$\sup_z |\ell(h_S, z) - \ell(h_{S\setminus i}, z)| \leq \gamma < \infty$$
>
> *is true for any training set S.*

The proof is too involved to be included in this chapter, but is a nice example of how concentration inequalities are useful in theoretical machine learning. The property involving γ is called *uniform stability* and simply states that the algorithm does no produce very different errors on unseen points z by merely changing one sample i of the training set.

The LOO bound shows that if the algorithm has uniform stability with a fast decaying γ, for example in $O(1/n)$, then the LOO error gets close to the true risk quickly, making it almost useful in practice. Almost because the cost of computing the LOO error for large scale datasets becomes prohibitive.

1.3.3 Test set

If we use cross-validation to perform model selection, then the performances of the resulting model are contaminated because we selected the best performing model on the validation data and the obtained error may thus not be a fair estimate of the true risk. This feedback from the data to the model is exactly the same reason that had us use a validation set to perform model selection. As such, we have exactly the same solution for obtained better estimate of the true error, which is collecting a third dataset, which we call *test*.

The proper way to perform machine learning is thus composed of three distinct dataset: a ***training*** set, a ***validation*** set and a ***test*** set. The training set is consumed by the ERM algorithm to select the best predictor from the family. The validation set is consumed by the model selection procedure that selects the right learning algorithm and its parameters (such as its computation budget, the number of data to train on, *etc* – these are called ***hyper-parameters***). The test set is solely used to get an idea of the real-world performances of the resulting model. Failure in following these clean rules and trying to use data for more than one purpose may result in ***overfitting***, that is, having a model that appears to perform well because its empirical risk on the training set (or the validation set) is low, but that performs badly in reality.

1.4 Nearest Neighbors

After having introduced the main formalism, we now present the ***Nearest Neighbors*** family of machine learning algorithms that relies only on a locality argument. Given pairs (x, y) with $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ and with the goal of predicting y from x, we assume that $\mathcal{X}$ is structured in a way such that nearby x have the same y.

The only practical requirement that this assumption raises is that $\mathcal{X}$ is endowed with a distance $\|x - x'\|$ that can be used to order element by proximity.

1.4.1 1-Nearest Neighbor

The simplest version of such algorithm is the first nearest neighbor, or 1-NN for short. Given a training set $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$, the prediction for any sample x is simply the y associated to the nearest x_i in $\mathcal{A}$. We provide below the pseudo-code of the algorithm. A python implementation is listed in ??.

Algorithm 1.3 Nearest Neighbor prediction

Input: Training set $\mathcal{A} = \{(x_i, y_i)\}_{i \leq n}$, sample x
Output: Prediction $\hat{y}$

1 Compute distance $\mathbf{d} = [\|x - x_1\|, \dots, \|x - x_n\|]$
2 Find nearest neighbor (x_i, y_i) such that $i = \arg\min_j \mathbf{d}_j$
3 **return** $\hat{y} \leftarrow y_i$.

What is the learning part of the first nearest neighbor? The answer is simply the memorization of the entire training set. In some sense, 1-NN is a very special learning algorithm because instead of condensing the information contained in the training set into a model, the model is the training set.

We show on Figure 1.1 the result of the 1-NN on a training set composed of 2-dimensional vectors as input and a binary label as output. The prediction associated to any 2d coordinate is shown in purple for one class and orange for the other. The training set is shown as circles of the corresponding colors and a validation set is shown as triangles of the corresponding colors. As we can see, the decision boundary (coordinates where the 1-NN switches from one class to the other) is a collection of straight lines. In fact, it is easy to prove that the decision regions correspond to the Voronoi cells of the training set, colored by their class.

Although the 1-NN rule dates as far back in time as the Persian empire according to Pelillo (2014), it has been formalized much more recently by Cover and Hart (1967). In the original paper, the

Listing 1.1 Vectorized 1-nearest neighbor prediction in NumPy following `scikit-learn` interface., label=lst:1nn

```
import numpy as np

class FirstNearestNeighbor:
    def fit(self, X, y):
        self.X = X
        self.y = y
    def predict(self, x):
        dist = ((self.X[None,:,:] - x[:,None,:])**2).sum(axis=2) # broadcast to B x n
            x dim
        index = np.argmin(dist, axis=1)
        return self.y[index]
```

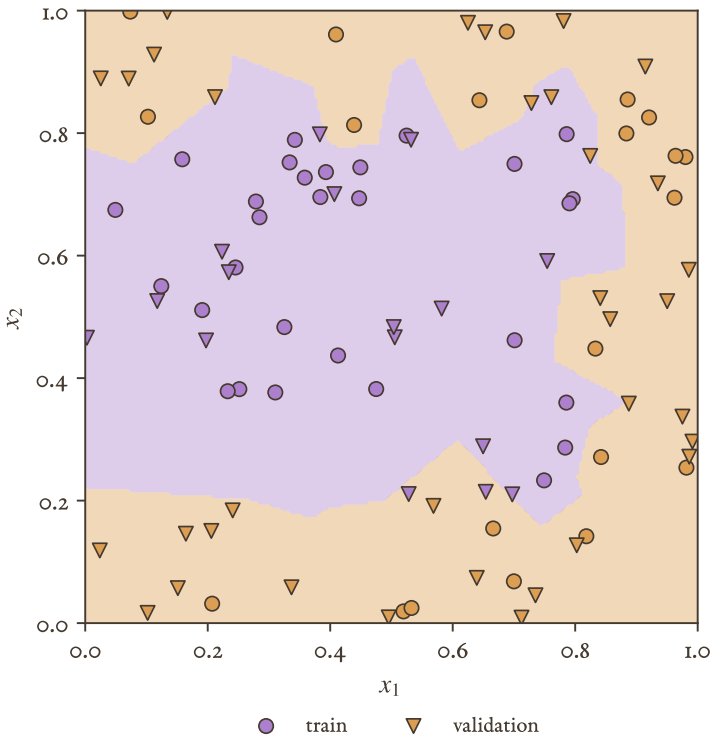

Figure 1.1 1-nearest neighbor decision regions on the synthetic dataset, with training and validation points overlaid.

authors proved a generalization bound for it. It requires the following lemma:

> LEMMA 1.14 (Nearest neighbor convergence)
> *Lemma: Let X be a metric space and p be such that with probability 1, x is either a continuity point of p with $p(x) > 0$ or a point on a non-zero probability measure. Let x'_n denote the nearest neighbor of x in the set $\{x_0, \dots, x_n\}$ sampled from p, then $x'_n \to_{n\to\infty} x$ with probability one.*

Proof. Let $\varepsilon > 0$, and consider the ball $B_\varepsilon(x)$ of radius ε centered on x. Since the x_n are sampled i.i.d. from p, then $P(\|x'_n - x\| > \varepsilon) = (1 - p(B_\varepsilon(x)))^n$. But $p(B_\varepsilon(x)) > 0$ by definition of x. Thus $P(\|x'_n - x\| > \varepsilon) \to_{n\to\infty} 0$. Furthermore, since $\|x'_n - x\| \geq 0$ is a non-increasing sequence, then $0 \leq \lim_{n\to\infty} \|x'_n - x\| < \infty$ almost surely. Yet, a.s. convergence implies convergence in probability and since convergence in probability are a.s. unique, combining with the previous results holds that the limit is 0 almost surely. ■

The generalization bound is asymptotic with respect to the number of training samples:

> THEOREM 1.15 (1-NN generalization Cover and Hart (1967))
> *Let X be a metric space. Let p_1 and p_2 be such that with probability 1, x is either 1) a continuity point of p_1 and p_2, or 2) a point on non-zero probability measure. Then in the limit $n \to \infty$ the* nearest neighbor true risk $\mathcal{R}$ *has the bounds*
>
> $$\mathcal{R}^\star \leq \mathcal{R} \leq 2\mathcal{R}^\star(1 - \mathcal{R}^\star),$$
>
> *with $\mathcal{R}^\star$ the Bayes error (irreducible error) $\mathcal{R}^\star = \mathbb{E}[\min_j \sum_i p_i(x)\ell(i, j)]$.*

Proof. The pointwise error is given by

$$r(x, x'_n) = P(y = 1|x)P(y' = 2|x'_n) + P(y = 2|x)P(y' = 1|x'_n) = p_1(x)p_2(x'_n) + p_2(x)p_1(x'_n)$$

and by Lemma 1.14 and using continuity of p_1, p_2 at x, we have

$$r(x) = \lim_{n\to\infty} r(x, x'_n) = 2p_1(x)(1 - p_1(x))$$

almost surely. Since the Bayes pointwise error is $r^\star(x) = \min\{p_1(x), 1 - p_1(x)\}$, we have that $r(x) = 2r^\star(x)(1 - r^\star(x))$ is almost sure. Taking the expectation over x leads to

$$\mathcal{R} = \mathbb{E}\left[2r^\star(x)(1 - r^\star(x))\right],$$

where the expectation and the limit can be swapped (dominated convergence) because $r(x'_n, x) \in [0, 1]$. We can bound from below by noticing that

$$\begin{aligned}\mathcal{R} &= \mathbb{E}[r^\star(x)] + \mathbb{E}[r^\star(x)(1 - 2r^\star(x))] \\ &\geq \mathbb{E}[r^\star(x)] \\ &\geq \mathcal{R}^\star,\end{aligned}$$

since $r^\star(x) \leq 1/2$.

We can bound from above by noticing that

$$\begin{aligned}\mathcal{R} &= 2\mathcal{R}^\star - 2\mathbb{E}[r^\star(x)^2] \\ &= 2\mathcal{R}^\star - 2\left(\mathbb{E}[r^\star(x)]^2 + \mathrm{Var}(r^\star(x))\right) \\ &\leq 2\mathcal{R}^\star(1-\mathcal{R}^\star),\end{aligned}$$

since the variance is always non-negative. ■

An interesting observation with Theorem 1.15 is that in the case of deterministic processes (*i.e.*, $\mathcal{R}^\star = 0$), the true risk is asymptotically null and the nearest-neighbor rule becomes a perfect prediction. This should make some intuitive sense: if the universe is perfectly deterministic and we have memorized all the possible states of that universe, then indeed predicting is the same as recalling and gives perfect answers. This link between learning and memorizing will come back in chapter 3 when we study what are known as *representer theorems*.

1.4.2 *k*-Nearest Neighbor

We can extend the nearest neighbor rule by considering k neighbors instead of only the first one. Intuitively, if the generating process is non-deterministic or if the boundaries between the different classes are jagged, considering more neighbors should smooth the boundary prediction by being more robust to the change of a single label in the neighbor set.

The pseudo-code of the k-Nearest Neighbor algorithm is given below:

Algorithm 1.4 k-Nearest Neighbor prediction

Input: Training set $\mathcal{A} = \{(x_i, y_i)\}_{i\leq n}$, parameter k, sample x
Output: Prediction $\hat{y}$

1 Compute distance $\mathbf{d} = [\|x - x_1\|, \dots, \|x - x_n\|]$
2 Find permutation π such that $\mathbf{d}_{\pi_1} \leq \mathbf{d}_{\pi_2} \leq \cdots \leq \mathbf{d}_{\pi_n}$
3 Gather nearest neighbor set $\mathcal{N} = \{(x_{\pi_i}, y_{\pi_i})\}_{i\leq k}$
4 **return** $\hat{y} \leftarrow \arg\max_j \sum_{(x_i,y_i)\in\mathcal{N}} \delta_{y_i,j}$ *with* $\delta_{i,j} = 1$ *iff* $i = j$, *else* 0 *(Kronecker).*

We show the effect of widening k in Figure 1.2. When increasing the number of neighbors, training samples that are isolated no longer contribute to the decision boundary. In our extreme case of $k = 15$, and because our training set is so small, this results in the entire left half-space being dominated by one class, even if there are samples of that other class. They are just not numerous enough to overcome the voting decision.

This begs the question of how to choose k so as to get a good predictor. Evidently, the *ERM* is not the proper way here, first because it compares algorithms of different families which it was not designed to do, and second because in practice $k = 1$ always has an empirical risk of 0, leading to it being among the best candidates every time.

The proper way to select k is thus to use cross-validation. We perform 5-fold cross-validation of k on the previous dataset and show the corresponding errors on train, val and test in Figure 1.3.

We notice that among the lowest average cross-validation error, we have $k = 1$ and $k = 21$, and both have similar error bars, indicating that they are equivalent predictors, at least according to our empirical study. Of course, they have different performances on the test set, but this should never

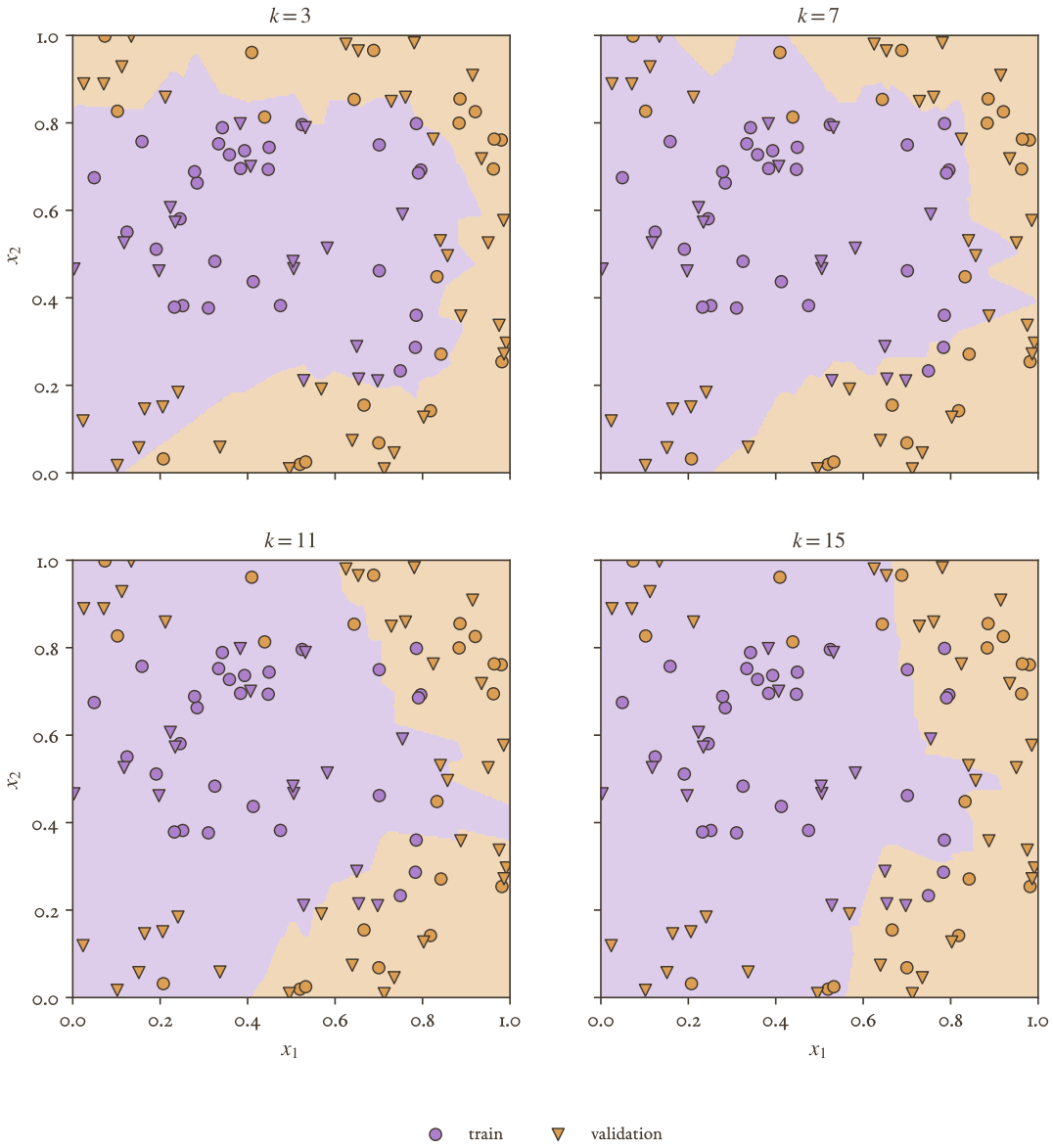

Figure 1.2 k-nearest neighbor decision regions on the synthetic dataset for $k \in \{3, 7, 11, 15\}$, with training and validation points overlaid. As k grows, the decision boundary smooths out.

be an argument for selecting $k = 1$ over $k = 21$. Indeed, model selection ***uses data, rendering their use impossible for further analysis*** and so the choice of k should be made ***before*** looking at the test performances. A good argument for selecting $k = 1$ is that it is a simpler rule and it is faster to compute. A good argument for selecting $k = 21$ is that it is more robust to outliers, if we know that our learning problem contains these.

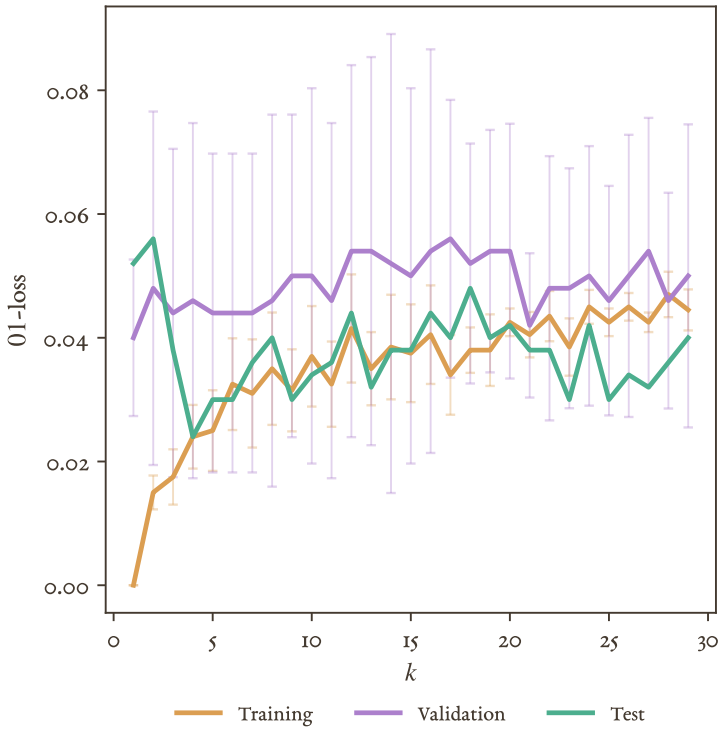

Figure 1.3 Searching for a good parameter k using 5-fold cross-validation on the synthetic dataset. The error bars are computed over the 5 splits of the cross-validation.

1.5 Learning Theory

As a conclusion for this first chapter, we briefly present some introductory words for some of the (many) learning theories that exist. The goal is not to be thorough or precise, but only to give an overview of the ***flavors of learning theory*** for the interested reader, since the remaining chapters will solely focus on existing algorithms.

1.5.1 Statistical/Probabilistic theory

The most popular view of machine learning is to fundamentally consider the problem of prediction as a stochastic process and then ask the question of whether we can find an algorithm that can guarantee a specified error rate in probability. This is the view introduced by Valiant (1984) in his seminal paper. The theory defines the concept of Probably Approximately Correct (PAC) and characterises what predictors are PAC-learnable. More formally, the definition of PAC-learnability is the following:

> **Definition 1.16** (PAC-Learnability)
> A predictor family $\mathcal{H}$ of mappings $\mathcal{X} \mapsto \mathcal{Y}$ is said to be PAC-learnable if for every $\varepsilon > 0, \delta > 0$, and every distribution $\mathcal{D}(\mathcal{X}, \mathcal{Y})$, there exists an algorithm A and a sample complexity $m(\varepsilon, \delta, \mathcal{D})$

such that running A on a training set $\mathcal{A}$ of $m(\varepsilon, \delta, \mathcal{D})$ samples from $\mathcal{D}$ returns a predictor h such that

$$\mathcal{R}_h \leq \mathcal{R}_{h^*} + \varepsilon,$$

with probability $1 - \delta$, where h^* is the best possible predictor from $\mathcal{H}$, *i.e.*, $h^* = \arg\min_{h \in \mathcal{H}} \mathcal{R}_h$.

The questions tackled by PAC learning theory are then ***what*** is PAC-learnable, and ***how*** learnable is it. The first searches for problem that can be successfully learned, the second optimizes the trade-off between the rates and the sample complexity.

We provide below a shortened version of the fundamental theorem of PAC learning, simplified from (Shalev-Shwartz and Ben-David, 2014).

THEOREM 1.17 (Fundamental theorem of PAC learning - simplified)
Let $\mathcal{H}$ be a hypothesis class over $\mathcal{X} \to \{0, 1\}$ and using the 01-loss function, then the following are equivalent

1. *$\mathcal{H}$ has the uniform convergence property*
2. *$\mathcal{H}$ is PAC learnable*
3. *Any ERM rule is a successful PAC learner for $\mathcal{H}$*
4. *$\mathcal{H}$ has finite VC dimension*

We will not prove the equivalences as it is too complex for this introductory chapter, and also because we have not yet provided the definition of the VC dimension (in chapter 3) or uniform convergence. We provide below the definition of uniform convergence for a predictor class:

DEFINITION 1.18 (Uniform convergence)
A predictor family is said to have the uniform convergence property if there exist $m_{\mathcal{H}} : (0, 1) \mapsto \mathbb{N}$ such that for every $\varepsilon > 0, \delta > 0$ and for every distribution $\mathcal{D}$, if a set $\mathcal{A}$, sampled i.i.d. from $\mathcal{D}$, contains at least $m_{\mathcal{H}}(\varepsilon, \delta)$ elements, then with probability at least $1 - \delta$

$$\forall h \in \mathcal{H}, |\hat{\mathcal{R}}_{\mathcal{A}}(h) - \mathcal{R}(h)| \leq \varepsilon.$$

Proof. Uniform convergence is better than sample complexity in that it does not involve the ERM, and as such the implications $1 \to 2$ and $1 \to 3$ are immediate. The implications $2 \to 4$ and $3 \to 4$ requires a no-free-lunch theorem we will not present. The implication $4 \to 1$ is proven in the one-side case in Corollary 3.6 and the two-sided version only doubles δ, but the implication also holds for the much stronger property that $\mathcal{H}$ is finite (which is not implied by 2 or 3, but implies finite VC dimension, see Proposition 3.5). In that case, we can use Hoeffding's inequality to the probability that any $h \in \mathcal{H}$ has a deviation of more than ε between the empirical risk and the true risk of the 01-loss by $2e^{-2m\varepsilon^2}$. Using the same union bound as we used for Theorem 1.12 and summing over all predictors gives us that $\mathcal{H}$ has uniform convergence with probability $\delta = 2|\mathcal{H}|e^{-2m\varepsilon^2}$. ■

1.5.2 *Algorithmic learning theory*

Before PAC, E. M. Gold proposed theory of computational learning in the landmark paper (Gold, 1967). In this paper, the question is whether a computer can perfectly uncover the mechanism underlying a process by only looking at examples. The setup is the following: suppose you have an alphabet and there exists a language class on this alphabet. Suppose additionally that an unknown

language from that class has been selected together with a method for providing information about it. The question of learnability then becomes whether there exists an algorithm that, if repeatedly presented with information about the unknown chosen language, makes only the same correct guess after some time.

The differences with the PAC theory are twofold: First, learning is considered successful here if it is absolutely correct. Second, the result is identified in the limit, that is, the sequence of guesses has to converge but nothing is said about the speed at which it converges. This raises a theoretical computer science question more than an engineering one that we could summarize by ***"is the identification of a complex process from observations about it alone, computable?"*** Gold proves that identifying more structured classes of languages requires more information, and that just having access to simple examples is not sufficient for more structured language.

Another question raised by algorithmic learning theory, or more precisely a closely related field of computational learning theory, is the complexity of the learning process. Indeed, even if learning is possible in the limit, it may be that the time or the memory to do so grows at a rate that makes it impractical.

1.5.3 Geometric theory

Finally, in a recent paper, Raginsky and Recht (2026) proposed a geometric interpretation of the concept of generalization. This interpretation comes after deep learning and generative AI have dramatically changed the field of machine learning. In particular, the leading companies are training what is called ***frontier models*** on a scale of data so unfathomable that the concept of unseen data becomes questionable. The main assumption is then ***There is no data distribution, only datasets*** and in fact this may well be true in many cases! Take for example a satellite that acquires images of the Earth. The satellite has a limited lifespan, and thus the number of images that it will acquire is finite. The learning problem may be how to train a predictor on the set of past observations such that it performs well on the finite set of future observations, but there is no distribution (besides the empirical distribution).

Formalizing that example, let us consider a training dataset $\mathcal{A}$ and a test dataset $\mathcal{B}$, and ask the question of how a predictor found by running an algorithm on $\mathcal{A}$ would perform on $\mathcal{B}$ depending on how $\mathcal{A}$ and $\mathcal{B}$ relate to one another. The generalization bounds investigated in the paper are thus of the form

$$\hat{\mathcal{R}}_{\mathcal{B}}(h) \leq \hat{\mathcal{R}}_{\mathcal{A}}(h) + D(\mathcal{A}, \mathcal{B}),$$

with D a dissimilarity between the two datasets.

The crux is to find predictor classes for which there is a tight and computable D.

1.6 Exercises

Exercise 1.1 — ERM failure example. Construct a toy hypothesis class and dataset where the empirical risk is zero but the true risk is large. Make the overfitting gap explicit and explain which assumption of ERM it violates.

Exercise 1.2 — k-fold cross-validation variance. Run k-fold cross-validation with $k = 2, 5, 10$, and leave-one-out on the same small 2D dataset and compare the variance observed.

2

Linear Models

In this chapter, we investigate the family of predictor that are linear in some feature space $\mathcal{X}$. More formally, given a sample $x \in \mathcal{X}$, then the prediction of a linear predictor $h \in \mathcal{H}$ can be written as

$$h(x) = \langle w, x \rangle,$$

which implies that $\mathcal{X}$ is an inner product space.

We also introduce the dataset that is used to support the course. It is called the ***bluebell dataset*** and is composed of 1800 images split into 12 classes of flowers commonly found in France, that were manually captured in the ***forêt de Saint-Germain en Laye*** near Paris. Figure 2.1 shows examples of each class. The images are centered and downscaled to 64×64 pixels to make working with raw images possible on a lightweight machine.

2.1 Linear prediction

We first start by considering several learning problems involving linear predictors through the lens of their risk and the corresponding loss function.

2.1.1 Principal Component Analysis

Given a space $\mathcal{X}$ from which we can sample observations, a first task that we can ask is whether ***there exists a subspace of $\mathcal{X}$ that contains almost all the information about any $x \in \mathcal{X}$***. In other words, we want to find a linear projection on $\mathcal{X}$, that compresses (***i.e.***, the output has less dimension than the input), and for which there exists a reverse projection such that the reconstructed sample is a good approximation of the original sample. Without loss of generality, we will constrain ourselves to linear

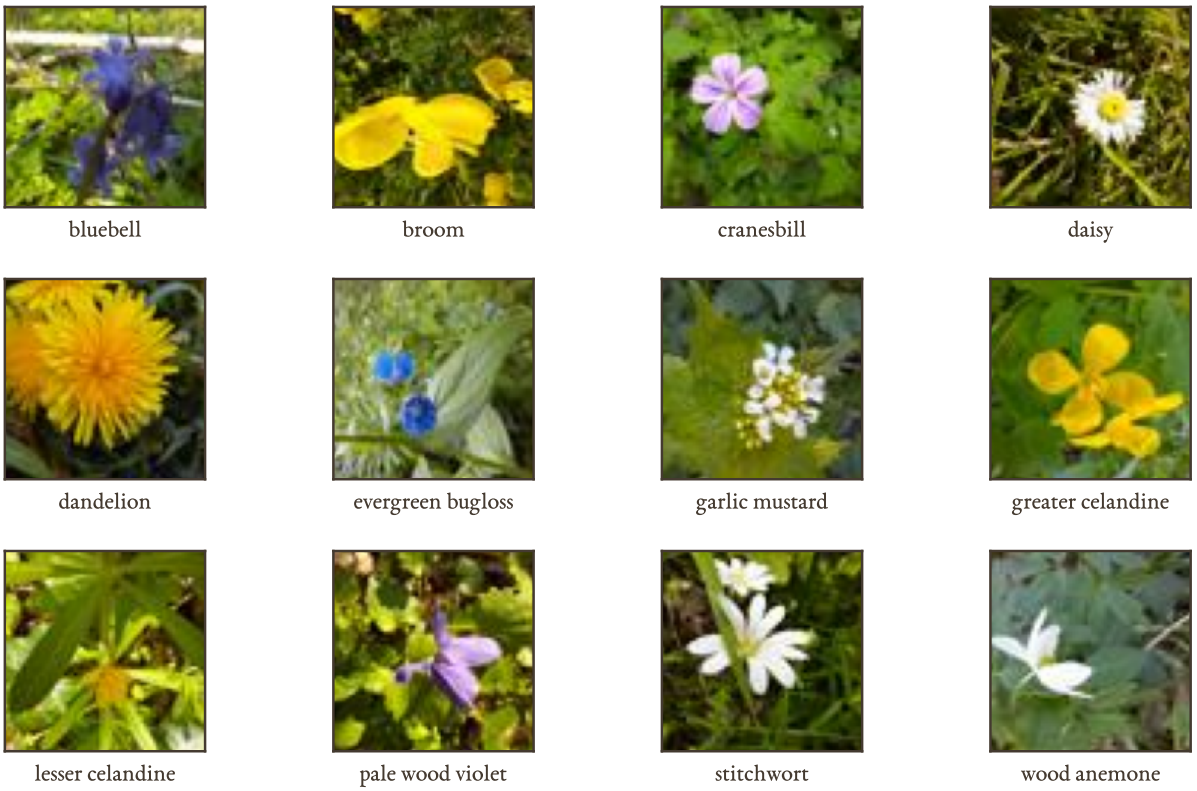

Figure 2.1 Examples of each of the 12 classes of the Bluebell dataset.

projections of target dimension 1, since a recursive application on successive residuals allows us to easily generalize to higher target dimensions. More formally and to be more specific, given a training set $\mathcal{A} = \{x_i \in \mathbb{R}^d\}_{1 \leq i \leq n}$, we want to find a linear predictor $w \in \mathbb{R}^{d \times 1}$ that minimizes the empirical reconstruction risk

$$\mathcal{L}(w) = \frac{1}{n} \sum_i \|x_i - ww^\top x_i\|^2, \tag{2.1}$$

with $\|w\|^2 = 1$.

After a bit of algebra, we can show that this is equivalent to the following maximization problem:

$$\max w^\top \Sigma w, \text{ s.t. } \|w\|^2 = 1,$$

with $\Sigma = \sum_i x_i x_i^\top / n$ and for which the solution is given by the eigenvector of Σ associated with the highest eigenvalue.

Proof. We compute the Lagrangian of the problem

$$\mathcal{L} = w^\top \Sigma w - \lambda(\|w\|^2 - 1),$$

and set its derivative with respect to the primal variable to 0:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w} &= 2\Sigma w - 2\lambda w \\ &= 0. \end{aligned}$$

This is an eigenvalue problem, so the solution is an eigenvector. Furthermore, the value of the objective function is equal to the eigenvalue and thus the eigenvector associated with the largest eigenvalue is the solution.

Alternatively, write the change of variable $u = w/\|w\|$ and the problem becomes

$$\max_u \frac{u^\top \Sigma u}{u^\top u},$$

which we recognize as the Rayleigh quotient (Σ is real and symmetric) and that is known to take its maximum for the eigenvector associated with the largest eigenvalue. ■

In practice however, doing so often reconstructs the average of the training set if the samples are far from the origin. If we remove the average from $\mathcal{A}$ before optimizing w, Σ becomes the empirical estimate of the covariance matrix on $\mathcal{A}$. The problem then becomes equivalent to searching for the projection that maximizes the variance in the output space and is known as ***Principal Component Analysis*** or PCA for short. More formally,

$$\max_w w^\top \sum_{i=1}^n (x_i - \mu)(x_i - \mu)^\top w, \text{ s.t. } \mu = \frac{1}{n}\sum_{i=1}^n x_i, \|w\| = 1.$$

Using the same arguments as before, we can show that the solution is the eigenvector with the largest eigenvalue. For the generic problem of output dimension p, it is easy to see that the solution consists of the p eigenvectors associated with the largest eigenvalues by orthogonality, between the eigenvectors.

The pseudocode for the PCA is given below.

Algorithm 2.1 Principal Component Analysis

Input: Dataset $\mathcal{A} = \{x_i \in \mathbb{R}^d\}_{1 \leq i \leq n}$, output dimension p
Output: Mean μ, projection $w \in \mathbb{R}^{d \times p}$

$\mu \leftarrow \frac{1}{n}\sum_{i=1}^n x_i$
$\Sigma \leftarrow \sum_{i=1}^n (x_i - \mu)(x_i - \mu)^\top$
$U, L \leftarrow \text{eig}(\Sigma)$ ordered by decreasing value of L
return μ, $U_{1:p}$

The corresponding python code is shown in Listing 2.1, although most machine learning libraries like `sk-learn` have an optimized implementation of it.

We show the first 8 principal components of the Bluebell dataset in Figure 2.2. The principal components look like oscillating functions (1 and 2 modes on the first line, ***etc***), and this is a well known fact in image processing. It has to do with the frequencies present in natural images, for which the intensity decays with a power-law, which makes the best reconstruction tackles low frequencies before high frequencies. PCA can be seen as a form of linear feature learning: starting from the initial sample x (here, an image), project it into a lower dimensional space that is maybe more suitable for analysis because some of the noise (***i.e.***, components that have very low variance) has been removed.

Usually, PCA is presented as finding the projection that maximizes the variance in the output space. It is then easy to see that the norm of the projectors has to be fixed to avoid unbounded solutions. With the norm constraints, we have shown that this is equivalent to a reconstruction problem, which falls into the category of ***self-supervised*** learning problems. In such problems, the

Listing 2.1 Principal Component Analysis in NumPy following `scikit-learn` interface.

```python
import numpy as np

class PCA:
    def __init__(self, p=1):
        self.p = p
    def fit(self, X):
        self.mu = X.mean(axis=0)
        Xc = X - self.mu
        Sigma = Xc.T @ Xc
        L, U = np.linalg.eigh(Sigma)
        order = np.argsort(L)[::-1][:self.p]
        self.U = U[:, order]
        return self
    def transform(self, x):
        return (x - self.mu) @ self.U
    def inverse_transform(self, z):
        return z @ self.U.T + self.mu
```

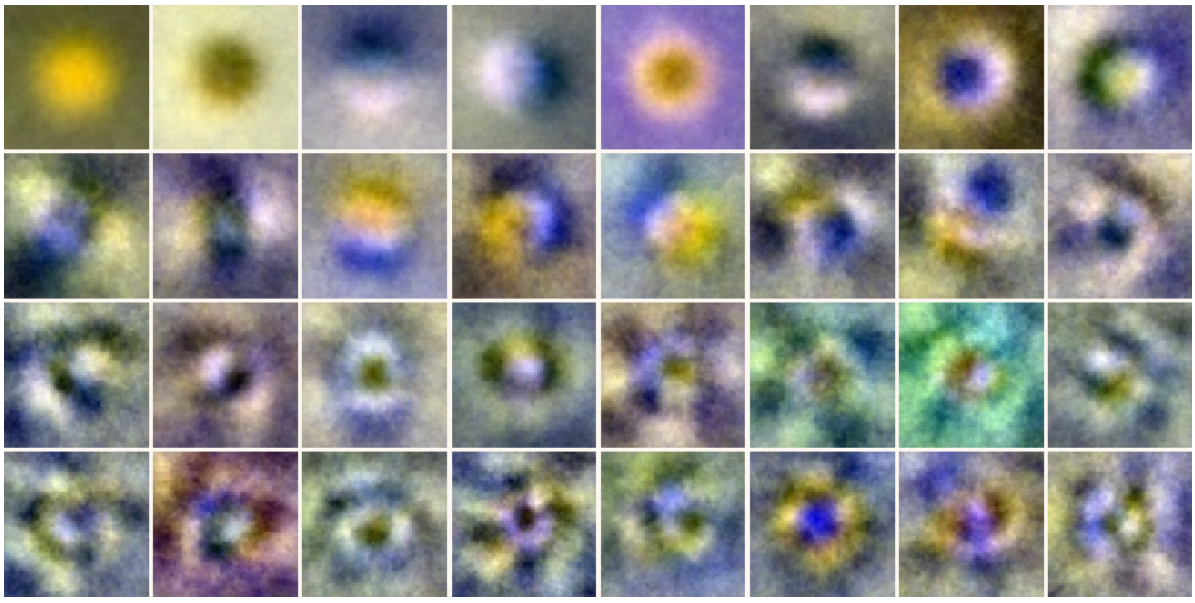

Figure 2.2 First 32 principal components of the Bluebell dataset, ordered by decreasing eigenvalue (left to right, top to bottom).

target y (here, $y = x$) is generated from the data rather than being an external label. In the simplest case, the target is the input, which leads to reconstruction tasks, and allows us to cast PCA as an ERM problem with the risk computed for the reconstruction loss.

One could ask what happens if we remove the norm constraint in the ERM formulation, as this is naturally bounded by the norm of the data. In that case, one can show that the solution has a norm constraint on w which leads to another eigenvalue problem and thus forces $\|w\| = 1$. This derivation is left as an exercise at the end of this chapter.

2.1.2 Linear regression

We are now interested is a ***supervised*** linear prediction problem where each sample x is associated with a target y. Taking example from the Bluebell dataset, we want a linear predictor that takes an image as input and outputs the index of its corresponding class.

More formally, given a training set $\mathcal{A} = \{(x_i, y_i)$ of pairs $x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$, the problem is set as minimizing the following empirical risk associated with the reconstruction loss:

$$\mathcal{L}(w) = \frac{1}{n} \sum_{i=1}^{n} \|y_i - w^\top x_i\|^2, \tag{2.2}$$

without any constraint on w.

This problem has a closed-form solution that is easy to obtain by setting the derivative of the loss with respect to w to 0:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{2}{n} \sum_{i=1}^{n} \left(x_i x_i^\top w - y_i x_i \right) = 0.$$

Then, provided that $\sum_{i=1}^{n} x_i x_i^\top$ is invertible, the solution is simply given by the pseudo inverse

$$w = \left(\sum_{i=1}^{n} x_i x_i^\top \right)^{-1} \left(\sum_{i=1}^{n} y_i x_i \right).$$

Alternatively, we can write the problem in matrix form to make the pseudo-inverse appear more readily. Let us denote X the matrix that stacks the x_i as rows and Y the column vector stacking the y_i accordingly. The risk can be rewritten as

$$\mathcal{L}(w) = \|Y - Xw\|^2,$$

and setting its derivative to 0 leads to

$$\frac{\partial \mathcal{L}}{\partial w} = 2(X^\top X w - X^\top Y) = 0.$$

Provided that $X^\top X$ is invertible, the solution is the Moore-Penrose pseudo-inverse

$$w = \left(X^\top X \right)^{-1} X^\top Y. \tag{2.3}$$

There are two issues with the pseudo-inverse solution. The first one is that $X^\top X$ has to be invertible which is the case only if it is full rank. If the n samples do not span the entirety of the d-dimensional space $\mathcal{X}$, or if they do but the last few dimensions are just noise and they make the inverse very much unstable, we can always resort to compressing them using PCA.

The second one is that computing the inverse can be extremely costly. Computing the matrix alone can be prohibitive for very large dimensions. In the case of our Bluebell dataset, the images are $64 \times 64 \times 3$ which means about 12 thousand dimensions, which should be feasible relatively easily on modern hardware, except on embedded devices with low memory and computing power. In the case of more realistic images having 1k pixels per side, the total dimension exceeds a million and storing the matrix in memory alone would require over a terabyte of fast access memory. In such case, one has to ditch the closed-form solution and opt for online optimization instead. The ERM problem can be solved by performing stochastic gradient descent, recalling that the gradient for a single sample is given by

$$\nabla \ell(w, x_i, y_i) = 2(w^\top x_i - y_i)x_i.$$

The pseudo-code for the stochastic gradient descent with mini batch is given below.

Algorithm 2.2 Linear regression using SGD

Input: Dataset $\mathcal{A} = \{(x_i \in \mathbb{R}^d, y_i \in \mathbb{R})\}_{1 \le i \le n}$, step-size schedule $(\eta_t)_{t \ge 0}$, mini-batch size B, horizon T

Output: Predictor w

$w_0 \leftarrow 0$
for $t \leftarrow 0$ **to** $T - 1$ **do**
  Sample $\mathcal{B} \subset \{1, \dots, n\}$ uniformly at random within $\mathcal{A}$, $|\mathcal{B}| = B$
  $g_t \leftarrow \frac{1}{B} \sum_{i \in \mathcal{B}} (w^\top x_i - y_i)x_i$
  $w_{t+1} \leftarrow w_t - \eta_t g_t$
return w_{T-1}

Applying this algorithm to the training and validation loss curves show in Figure 2.3. After 10k steps, the average train accuracy, measured as the agreement between the prediction quantized to the nearest integer and the label, is 10.1% and the average validation accuracy is 9.8%. This is not a good predictor. Guessing at random has a $1/12 = 8.3\%$ chance of being correct (and so is always answering 0), and our linear regression is barely better than that. There are several issues at play here. First and foremost, the type of flower contained in an image is definitely not a linear relation of its pixel intensities. Assuming linearity is a terrible model because it ignores many of the image properties, like shift invariance (***e.g.***, take the same image but remove the leftmost column and add an additional column to the right, the label does not change). Second, we arbitrarily structured the output space such that the labels are on the number line is a specific order. The regression loss penalizes errors that predict nearby labels less than those who predict faraway labels, whereas our task measures the accuracy regardless of the type of errors. This problem formulation that consists in regressing continuous values whereas the labels correspond to exclusive attributes is a bad setup. We insist on that point.

We can also see some signs of ***overfitting***, as the training loss continues to decrease whereas the validation loss plateaus. At this point, it becomes clear that running more iterations of SGD is going to have the model select each of the unique pixel combination that can produce the label for each each and memorize it, rather than uncovering some broader pattern.

A note worth mentioning: in our model, we chose to not add an additional bias, with which the predictor would take the form $w^\top x + b$, to simplify the presentation. This bias term (called ***intercept***

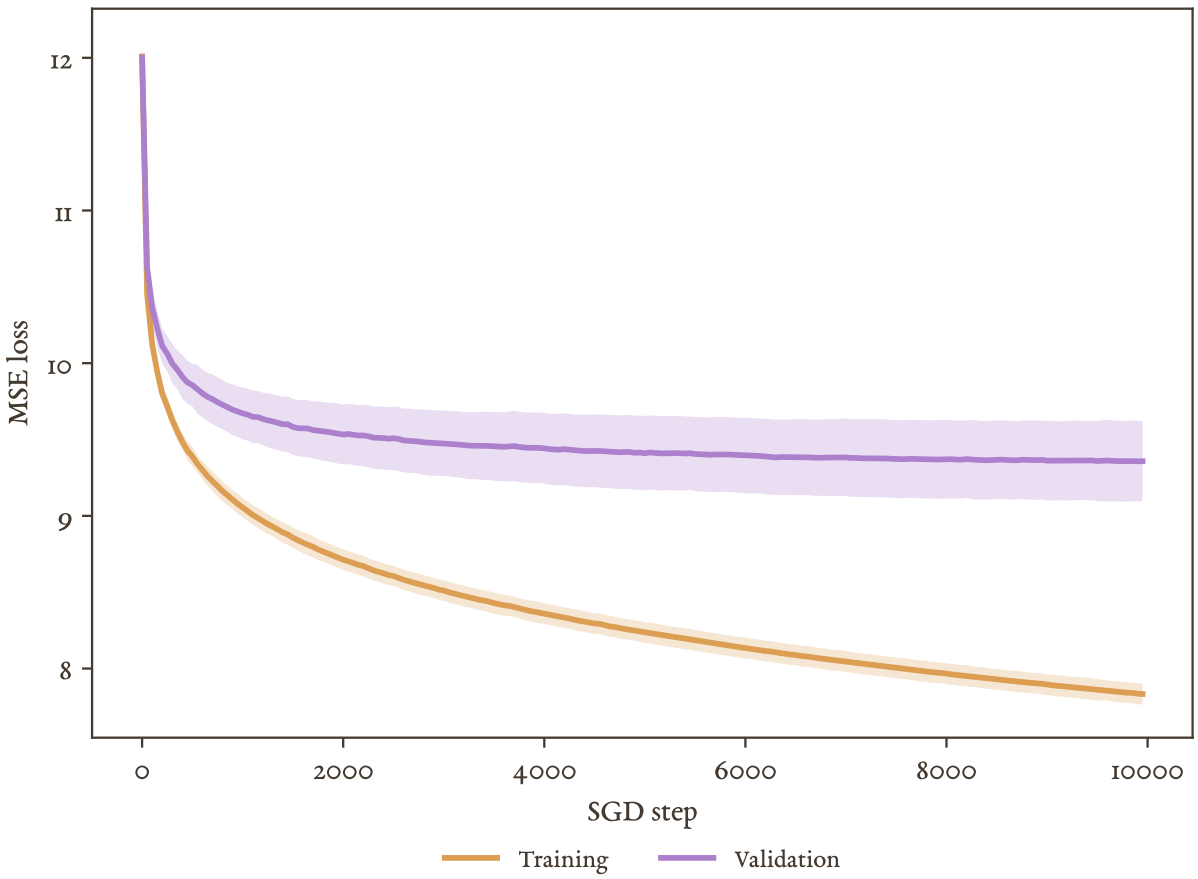

Figure 2.3 Training and validation loss curves for regressing the label index on Bluebell with SGD ($B = 32$ and $T = 10000$). 90% confidence interval over the 5-fold splits shown in pale colors.

in logistic regression – see below) can easily be implemented by concatenating a constant 1 to the input. It can serve as a calibration tool by setting the default prediction of the model which can be important in low dimensional cases or for problems with high imbalanced (some labels are much more probable than others). In very high dimension problems, it is often not very impactful, and can be dropped.

2.1.3 Linear classification

Because our Bluebell dataset is associated with a discrete set of labels corresponding to the type of flowers, the natural task to solve is a classification task. This can be effectively tackled by changing the loss used in our ERM algorithm to one that measure something closer to the definition of accuracy in a classification task. We show here two examples of such losses, leading to different frameworks.

Hinge Loss. Consider the binary case, in which there are only two possible labels types, which we call classes. The 01-loss in that case simply counts 1 when the predicted class and the true class disagree. By mapping the first class to 1 and the second class to -1, the sign of the product between the predicted class $\hat{y}$ and the true class y is sufficient to compute the 01-loss:

$$\ell_{01}(\hat{y}, y) = \max(0, -\text{sign}(\hat{y}y)).$$

Notice how this definition does not require $\hat{y}$ to be an integer value from $\{-1, 1\}$. Indeed, any real value is valid in the formula. From now on, we thus restrict only y to be in $\{-1, 1\}$, whereas $\hat{y} \in \mathbb{R}$.

Interestingly, any upper-bound of the 01-loss is guaranteed to also optimize it at least as good as the bound gap. In other words, using an upper-bound of the 01-loss in an ERM algorithm and achieving 0 empirical risk guarantees 0 empirical 01-loss also. The simplest of such upper bounds is the ***hinge*** loss, which is linear by part, and has the following definition:

Definition 2.1 (Hinge loss)
Given a predicted class $\hat{y}$ and a true class y, the hinge loss is defined as

$$\ell_{\text{hinge}}(\hat{y}, y) = \max(0, 1 - \hat{y}y).$$

Proposition 2.2 (Hinge loss bounding the 01-loss)
The hinge loss is an upper-bound of the 01*-loss.*

Proof. If the true class and the predicted class share the same sign, their product is positive and the hinge loss is between 0 and 1 whereas the 01-loss is 0. Else, the product is negative and the hinge loss is greater or equal to 1 whereas the 01-loss is 1. ■

The hinge loss and the 01-loss are shown on Figure 2.4. Notice that the hinge loss is convex. the part between 0 and 1 is the only one that deserves specific attention because the hinge loss counts an error whereas the 01-loss does not. This means that the hinge loss is more conservative and that it requires $|\hat{y}| \geq 1$ instead of just non-negativity.

Now that we know a loss that correctly models our classification problem, how do we train a linear predictor? The answer in the case of the Bluebell dataset is not as straightforward as in the binary classification case, because we have 12 different classes. The easiest way of taking that into account is to cast our 12-class problem as 12 binary class problems. We do this by considering one class against all others for each of the 12 possible classes. As such, the true label y becomes a vector

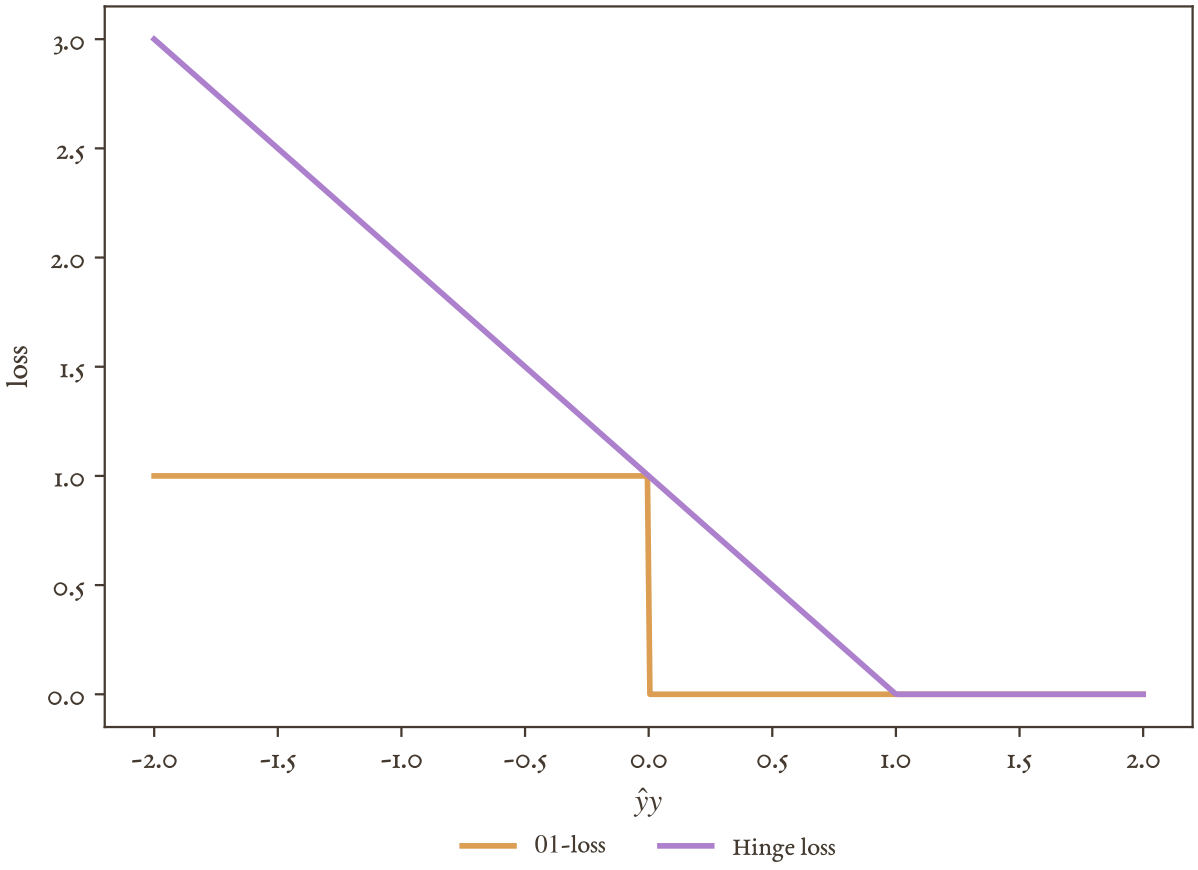

Figure 2.4 Graph of the 01-loss and the hinge loss.

of dimension 12 with 1 on the dimension corresponding to its true class, and −1 on all others. This approach is known as *One-against-All*.

Equipped with these new labels, we can consider a set of linear predictors $\{w_c\}$ for each of the $1 \leq c \leq 12$ classes. For each of them, we can solve the ERM problem associated with the following risk:

$$\mathcal{L}_c = \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(w_c^\top x_i, y_i[c]), \tag{2.4}$$

with $y_i[c]$ being the c component of y. Notice that since the hinge loss is convex and this problem is a sum of convex functions, it is convex. Each of these problems are independent and can be solved in parallel. Notice however, that this does not tell which class is the one predicted. Contrarily to a single predictor where the sign can easily decide for the predicted class, here, we have 12 real values to choose from. Because the hinge tries to *push* the prediction beyond an absolute value of 1, we can infer that the class with the largest value is the one to be the closest to the positive signal of exceeding 1. Thus, the natural rule is to take the arg max among the predictions.

Sadly, there is no closed-form solution to this linear classification problem. Instead, we have to rely on iterative optimization, which we can do by using SGD again. We alleviate the problem of the hinge loss not being differentiable at 0 by using a subgradient, for instance 0.

The code in pytorch for training a multiclass linear predictor with the hinge loss is show in Listing 2.2. Running it on the Bluebell dataset for $T = 10000$ steps leads to an average training accuracy of 66.4% and an average validation accuracy of 55.1%. Here, the accuracy is simply defined as the rate at which the arg max agrees with the true class. This is already much better than with the regression on the arbitrary classe indices and shows that the proper problem formulation is an important choice. The loss curve is shown in Figure 2.5. However, overfitting is still there, as shown by the gap between the training and validation accuracies.

Logistic regression. In the previous problem formulation, we did not take into account that the classes are exclusive, that is, that being of one class forbids being of any other. Our different predictors trained with the hinge loss were independent during training, and it is only during validation that we resorted to use the arg max for selecting the predicted class.

But is there a way to enforce a dependance between the predictions? An answer could lie in predicting the probability of each class being the true class. Since probabilities have to sum to 1, an increased probability on one class means lower probabilities on all others. Transforming our set of predictions to probabilities can be achieved using the *softmax* function.

> Definition 2.3 (Softmax)
> The softmax function is defined as
>
> $$\sigma : \mathbb{R}^d \to [0,1]^d$$
> $$\hat{y} \mapsto \left[\frac{e^{\hat{y}_i}}{\sum_{j=1}^{d} e^{\hat{y}_j}} \right]_{1 \leq i \leq d}.$$

Now that we have mapped $\hat{y}$ to probabilities, we need a loss function that penalizes high probabilities of non-occurring classes. The categorical cross entropy does exactly that.

Listing 2.2 Linear prediction with the hinge loss for multiple classes using pytorch following `scikit-learn` interface.

```
import torch

class SGDHinge:
    def __init__(self, n_classes, eta0=1e-3, batch_size=32, steps=10000):
        self.n_classes = n_classes
        self.eta0 = eta0
        self.batch_size = batch_size
        self.steps = steps
    def fit(self, X, y):
        X = torch.as_tensor(X, dtype=torch.float32)
        Y = 2 * torch.nn.functional.one_hot(torch.as_tensor(y, dtype=torch.long),
            self.n_classes).float() - 1
        n, d = X.shape
        self.w = torch.zeros(d, self.n_classes)
        for t in range(self.steps):
            eta = self.eta0 / (1 + t) ** 0.5
            idx = torch.randint(0, n, (self.batch_size,))
            Xb, Yb = X[idx], Y[idx]
            margin = (Xb @ self.w) * Yb
            g = -(Xb.T @ (Yb * (margin < 1))) / self.batch_size
            self.w -= eta * g
        return self
    def decision_function(self, x):
        return torch.as_tensor(x, dtype=torch.float32) @ self.w
    def predict(self, x):
        return self.decision_function(x).argmax(dim=1).numpy()
```

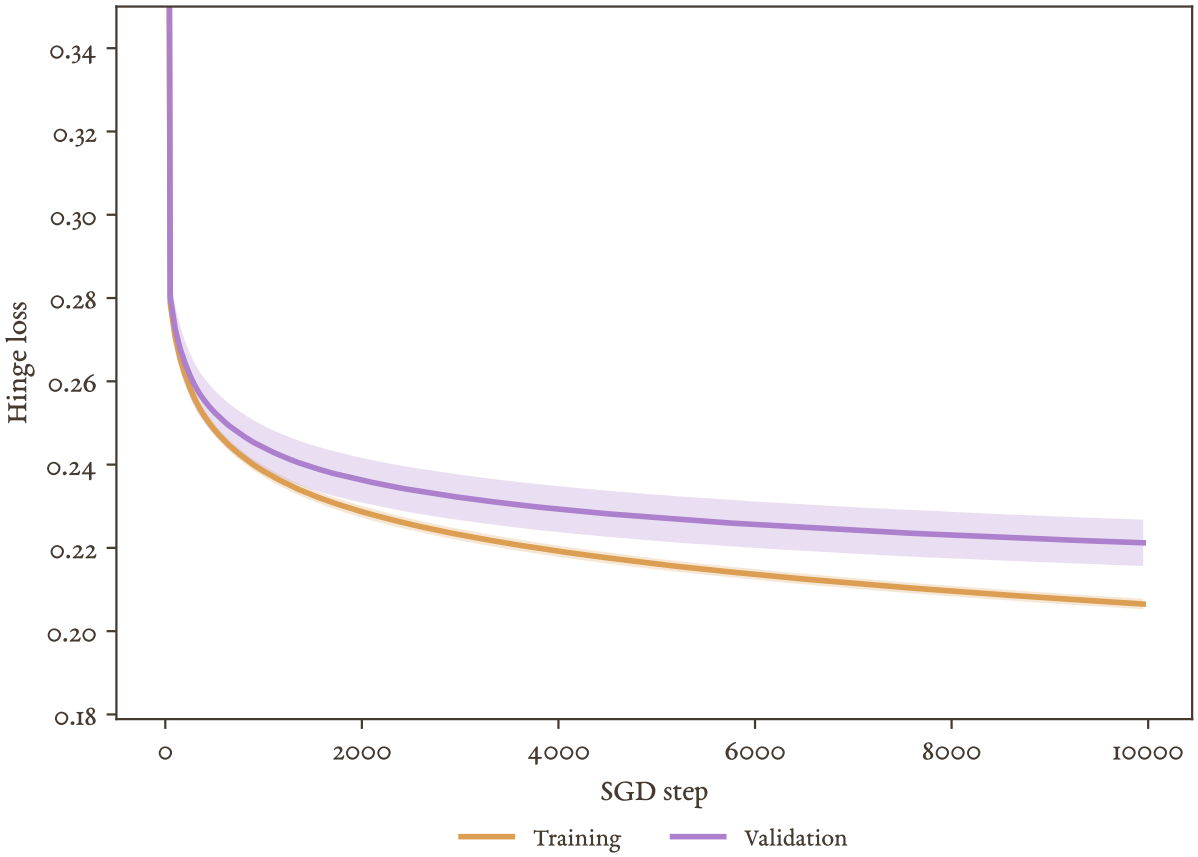

Figure 2.5 Training and validation loss curves for classification using the hinge loss on Bluebell with SGD ($B = 32$ and $T = 10000$). 90% confidence interval over the 5-fold splits shown in pale colors.

Definition 2.4 (Categorical cross-entropy)
Given a target prediction $\hat{y} \in \mathbb{R}^d$ and true probabilities $y \in [0,1]^d$, the categorical cross-entropy is

$$\ell_{\text{CCE}}(\hat{y}, y) = \sum_{i=1}^{d} -y[i] \ln(\sigma(\hat{y})[i]),$$

with $\sigma(\hat{y})[i]$ being the components of the softmax function applied to $\hat{y}$.

Notice how our definition of the categorical cross-entropy include the softmax function. This is an arbitrary choice and for models that already output probabilities, it can be silently dropped from the definition.

Optimizing a linear model with the categorical cross-entropy is known as ***logistic*** regression. The name comes from the non-linear function that in the binary case corresponds to a sigmoid, known in French as the ***logistique*** function. The ERM problem is simply minimizing the risk

$$\mathcal{L}_{\text{CCE}} = \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{CCE}}(W^\top x_i, y_i),$$

with W the matrix concatenating the w_c for all the classes $1 \leq c \leq 12$ of our Bluebell dataset. Solving this ERM problem can be done using SGD, which requires computing the gradient $\nabla_W \ell_{\text{CCE}}(W^\top x_i, y_i)$. This can be done by hand or better, using an automatic differentiation library to avoid the unnecessary errors.

The pytorch code for logistic regression is shown in Listing 2.3. Running logistic regression on the Bluebell dataset leads to an average training accuracy of 83.8% and average validation accuracy of 64.5%. This is a significant improvement over the independent model of the hinge loss, and a considerable achievement for a linear model when compared to the original 8.3% of the random predictor. However, there is still strong indication of overfitting given the gap between the training and validation accuracies.

2.2 Generalized linear models

In the case where there is doubt that a linear predictor is the right model for the problem, one can be tempted to transform the input such that it becomes the case. This is a powerful idea. Instead of being limited by linear relations between x and y, we propose to transform x using a non linear map $\phi(x)$ and then perform linear regression, with the hope that the mapped samples $\phi(x)$ have better linear relation to y. Doing so does not changed the regression problem at all since the mapping is done before the ERM algorithm.

We propose here to investigate two commonly used transforms, polynomials and sine functions.

2.2.1 Polynomial map

A simple way to enforce non-linear prediction with a linear predictor is to consider the map $\phi : x \mapsto [x, x^2, \dots, x^p]$ that can be applied element wise on any vector of dimension d to obtain a vector of dimension dp. With that mapping applied, the linear predictor then simply becomes

$$\langle w, \phi(x) \rangle.$$

Effectively, we have not changed the ERM problem, which can be solved with the same methods described previously.

Listing 2.3 Logistic regression using pytorch following `scikit-learn` interface.

```
import torch

class SGDLogisticRegression:
    def __init__(self, n_classes, eta0=1e-3, batch_size=32, steps=10000):
        self.n_classes = n_classes
        self.eta0 = eta0
        self.batch_size = batch_size
        self.steps = steps
    def fit(self, X, y):
        X = torch.as_tensor(X, dtype=torch.float32)
        y = torch.as_tensor(y, dtype=torch.long)
        n, d = X.shape
        self.w = torch.zeros(d, self.n_classes, requires_grad=True)
        for t in range(self.steps):
            eta = self.eta0 / (1 + t) ** 0.5
            idx = torch.randint(0, n, (self.batch_size,))
            Xb, yb = X[idx], y[idx]
            loss = torch.nn.functional.cross_entropy(Xb @ self.w, yb)
            loss.backward()
            with torch.no_grad():
                self.w -= eta * self.w.grad
            self.w.grad.zero_()
        return self
    def decision_function(self, x):
        return torch.as_tensor(x, dtype=torch.float32) @ self.w
    def predict(self, x):
        return self.decision_function(x).argmax(dim=1).detach().numpy()
```

To illustrate the effectiveness of that approach, let us consider a one-dimensional case where we try to model the unknown non-linear relation between a scalar x and a scalar y using linear prediction on a polynomial map. The true relation is itself a polynomial of degree 3, but the training set is composed of noisy observations. We show on Figure 2.6 the results for varying degree p from 1 to 7, as well as the coefficients for each power, compared to the true value.

The first thing we can notice is that, obviously, without the polynomial map, the prediction is very bad. Second, when the degree exceeds that of the true function, which is our proxy for simulating a map that is more complex than the phenomenon we are trying to predict, we can see that unnecessary powers have non-zero coefficients. This is explained by the need for the model to fit the noise, but also by numerical imprecision in the algorithm. Such model is called ***over-parametrized***, because it has too many parameters compared to the problem. These non-zero coefficients at higher degree do not seem to have much influence on the curve, but be not deceived by the appearance of the plot that is constrained in $[-1.5, 1.5]$ A polynomial of degree 7 is sure to be very different from one of degree 3 the further the input is from 0. The question then becomes how to choose the maximum degree. The simple answer for now is using cross-validation. We show the average training and validation error using cross-validation on Figure 2.7. Thankfully, cross-validation is able to constrain the choice of the degree to something reasonable between 3 and 5, whereas the training error continues to decrease. This is a very good example on why the training loss cannot be trusted for selecting hyper-parameters.

In the vector case, notice that the polynomial map we proposed does not take into account interactions between different components of the input, like $x[i]^k x[j]^l$ for some components i, j and degrees k, l. These interactions can be taken into account by considering tensor product of the vector x with itself. Informally, the tensor (outer) product can be defined as the operator $\otimes$ such that

$$\langle a \otimes a', b \otimes b' \rangle = \langle a, b \rangle \langle a', b' \rangle,$$

which by expanding the definition of the inner product $\langle a, b \rangle = \sum_i a[i]b[i]$ makes the inter-products visible.

For convenience of notation, we will write

$$x^{\otimes p} = x \otimes \cdots \otimes x$$

the tensor product of x with itself p times. such map contains all products of components of x such that the sum of the degrees is p. For $p = 2$, $x^{\otimes 2} = xx^\top$ which is a matrix. Degree $p = 3$ can be obtained by unrolling $x^{\otimes 2}$ into a vector and multiplying it with $x^\top$. Higher degrees are following the same procedure of unrolling into a vector and getting a matrix by multiplying with $x^\top$. Needless to say that such map is extremely memory intensive since the size of the resulting vector grows in d^p with d the original dimension of x. For now, such is the price to pay to have a very expressive model while doing only linear prediction. We will see in chapter 3 that the very definition of the tensor product can be leveraged to avoid that memory cost, at the expense of abandoning explicit linear modelling and paying a higher computational cost.

2.2.2 *Periodic maps*

If we have some a priori knowledge about the signal, for example that the input signal has periodic properties (***e.g.***, dates and the target is subject to seasonal changes), we can leverage it and construct an explicit map taking it into account. An easy example is the following sine map:

$$\phi : x \mapsto [\sin(f_0 x), \sin(2 f_0 x), \dots, \sin(p f_0 x)],$$

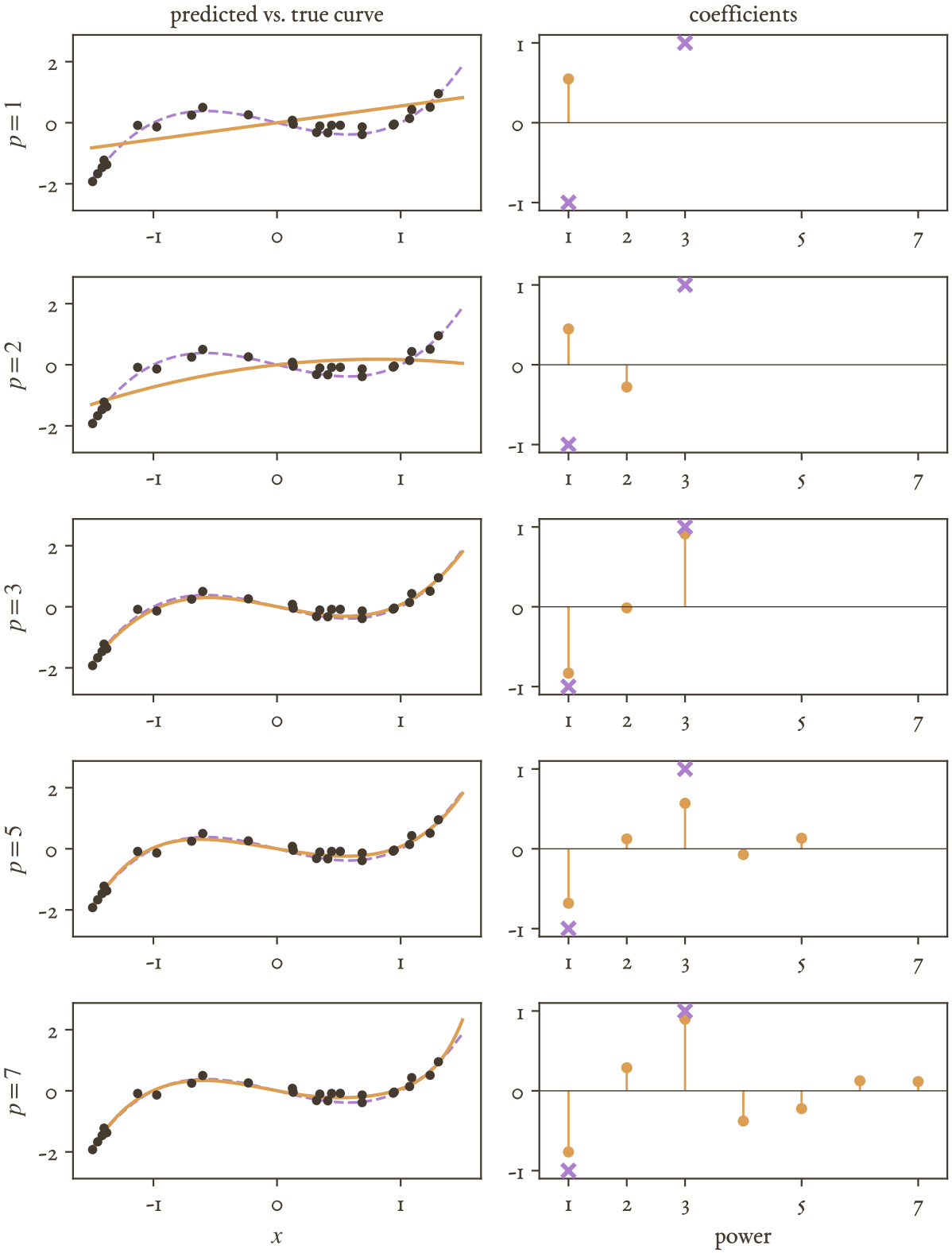

Figure 2.6 Generalized linear regression with a polynomial map. Left: predicted curve, true curve and training samples. Right: stems represents the value associated with each power up to the corresponding degree.

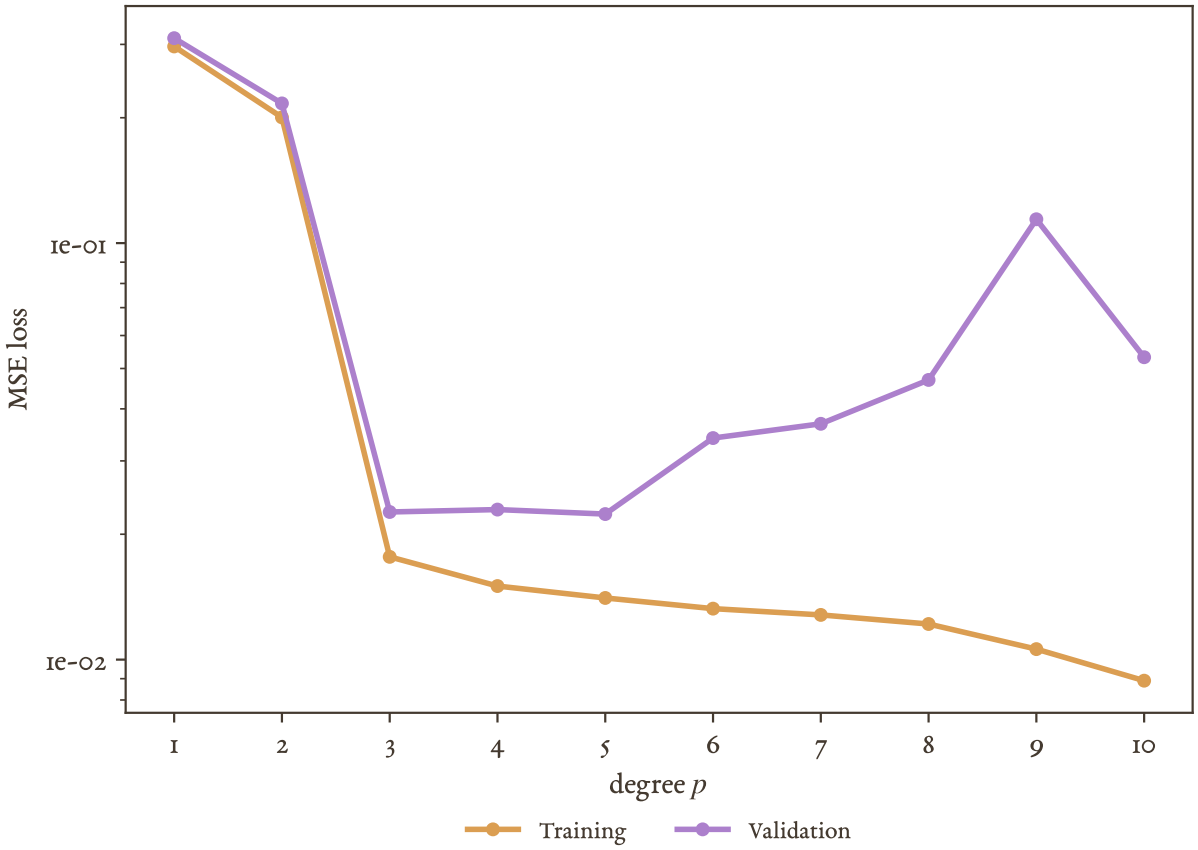

Figure 2.7 Training error and validation error for varying degree of a polynomial generalized linear regression using 5-fold cross-validation.

that has a set of p sine waves with base frequency f_0i, all applied component-wise.

Similarly to the polynomial map, this mapping can be extended to model periodicity in direction of the input space that are not necessarily following the axes. For that matter, let us consider a set of p random normal vectors $\{\omega_i \sim \mathcal{N}(0, \sigma) \in \mathbb{R}^d\}$ and define the map

$$\phi : x \mapsto [\sin(\omega_1^\top x), \cdots, \sin(\omega_p^\top x)].$$

To get a sense of what such maps encode, let us consider the inner product between two input vectors x and x',

$$\langle \phi(x), \phi(x') \rangle = \sum_{i=1}^{p} \sin(\omega_i^\top x) \sin(\omega_i^\top x').$$

Using trigonometric identities, we arrive to

$$\langle \phi(x), \phi(x') \rangle = \frac{1}{2} \sum_{i=1}^{p} [\cos(\omega_i^\top (x - x')) - \cos(\omega_i^\top (x + x'))],$$

which seems to use $x - x'$ and $x + x'$. In other words, such map is able to consider the difference between x and x'. Relying only on such difference is called ***shift invariance***, and the current map does not possess it because of the $x + x'$ term, but it can be fixed contrarily to the linear model (which corresponds to ϕ being the identity). The fix will be presented in chapter 3.

The choice of the number p of frequencies and the range σ of the frequencies to consider are highly influencial to the performances and have to be properly selected via cross-validation. the key difference with this model and the polynomial one, is that we could potentially sample an infinite number of ω_i and using cross-validation to select the good ones is at risk of overfitting on the validation set by focusing on the these samples.

2.3 Regularization

A better way of handling shortcomings of the predictor is to somehow ***observe*** it implicitly everywhere. This can be done by adding a structural cost that penalizes function that take values outside of the training samples that are radically different from the ones on the training examples. One easy way of doing so it to penalize functions that use all the parameters of their family, for example a high degree polynomial that has non-zero coefficient on all powers. Such function is the most complex of its family and should be attained only in rare cases where that capacity is required.

Let us consider noisy observations of a linear phenomenon,

$$y = \mathbf{a}^\top \mathrm{x} + \varepsilon, \varepsilon \sim \mathcal{N}(0, \sigma),$$

and assume that $\|\mathbf{a}\|_0 < d$ (not all input dimensions are used), can we force a linear predictor obtained through ERM to be also sparse? This would be a good way of getting rid of the problematic non-zero coefficients of our generalized linear models when they are over-parametrized. In the more general case than this simple example, are there ERM algorithms that do not use the entire capacity of the predictor family when it is unnecessary?

It turns out that all we need to do is to penalize complex solutions. This is achieved by adding a ***regularizer*** $\Omega(\cdot)$ that increases with the complexity of our predictor w. More formally, in the case of

linear prediction, we now try to minimize the augmented empirical risk

$$\mathcal{L}_\Omega(w) = \frac{1}{n}\sum_{i=1}^{p} \ell(w^\top x_i, y_i) + \Omega(w), \tag{2.5}$$

which can of course be generalized using a mapping ϕ on x_i.

We present here two of such regularizers that are very popular in the literature and in practice: the *LASSO* and the *Ridge*.

2.3.1 LASSO

If we have a sparse a priori on w, the natural regularizer would be $\|w\|_0$ as the ℓ_0 norm penalizes all non-zero components. However, it turns out that it is also extremely hard to optimize, because it leads to a combinatorial problem where one has to test all the possible combinations of active components.

To get a sparse prediction without increasing too much the hardness of the problem to solve, Tibshirani (1996) proposed the Least Absolute Shrinkage and Selection Operator, or LASSO for short, that consists in penalizing w with the ℓ_1 norm $\|w\|_1 = \sum_{i=1}^{d} |w[i]|$.

The regularized ERM problem consists then in minimizing the regularized risk

$$\mathcal{L}_\Omega(w) = \frac{1}{n}\sum_{i=1}^{p} \ell(w^\top x_i, y_i) + \lambda\|w\|_1,$$

which is convex if the loss is convex, since the norm ℓ_1 is convex. The hyper-parameter λ is used to control the trade-off between the empirical risk and the regularization, that is, between how much we want to fit the data ***versus*** how simple we want our solution to be.

Over the past decades, there have been many different algorithms to solve the LASSO, but for large scale problems, we can trust our good old stochastic gradient descent. In that case, the subgradient for the ℓ_1 norm is chosen to be 0 at 0. With this strategy, it is easy to see that a component will stop to receive gradient updates from the regularizer only if it is exactly 0, will probably never happen since that would require the step size to be exactly the value of the component. As such, an SGD strategy spends most steps having unnecessary components jittering around 0 while never attaining it, and a common post-processing is to threshold tiny values after training.

The main appeal for the LASSO is that enforcing a sparse model allows the practitioner to see which variables are in play. It has thus an explanatory value.

On Figure 2.8, we fit a degree 10 polynomial similar to the previous figure of subsection 2.2.1, and we show the number of components above 10^{-3} depending on λ, as well as the corresponding average training and validation loss. As we can see, too little regularization leads to using all the 10 degrees while too much regularization makes both the training and validation loss explode. Between those two extremes, there is a sweet spot where the validation loss does not increase significantly compared to the unregularized approach, but only roughly half the components are used, meaning the resulting model is less unnecessarily complex.

To understand why the LASSO produces sparse solutions, we can perform some analysis of the solution for the special case where x is projected into its eigenspace using its singular value decomposition. Writing X the matrix concatenating all the $x_i \in \mathcal{A}$ row-wise, and writing $X = VSU^\top$ its singular value decomposition, we consider the ERM problem associated with the following augmented risk

$$\mathcal{L}(w) = \|Y - XUw\|^2 + \lambda\|w\|_1,$$

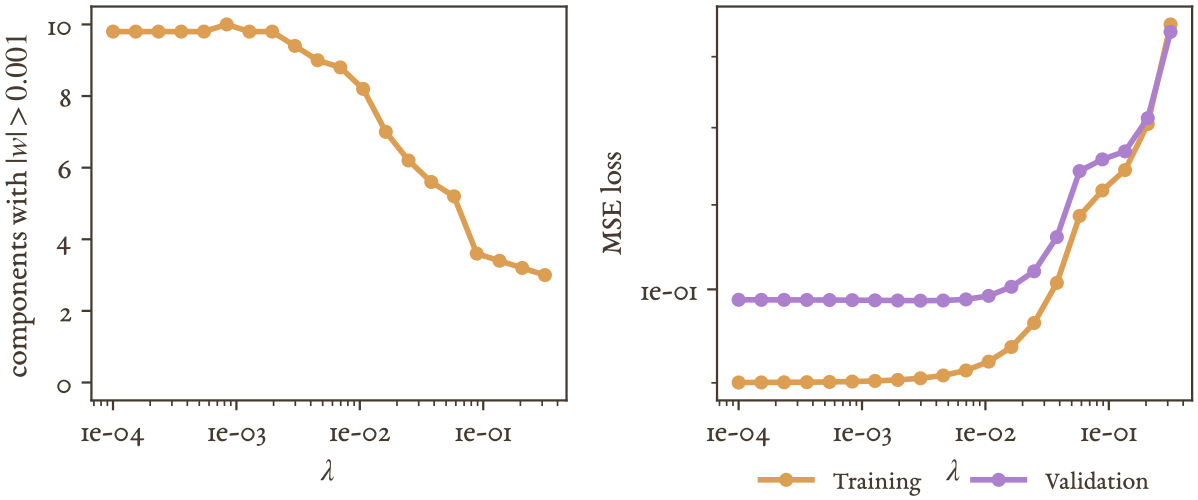

Figure 2.8 Training error and validation error for varying values of the λ hyper-parameter of the LASSO regularized linear regression using 5-fold cross-validation and degree 10 polynomials.

where X has been projected by U. We can see that

$$\mathcal{L}(w) = \|Y - VSw\|^2 + \lambda\|w\|_1,$$

since $U^\top U = I$. Setting the derivative of $\mathcal{L}$ to 0 gives

$$\frac{\partial \mathcal{L}}{\partial w} = 0 = -2SV^\top Y + 2S^2 w + \lambda \text{sign}(w),$$

leading to

$$w = S^{-1}V^\top Y - S^{-2}\frac{\lambda \text{sign}(w)}{2}.$$

Let $\tilde{w} = S^{-1}V^\top Y$ and note that $\text{sign}(w) = \text{sign}(\tilde{w}) = \frac{\tilde{w}}{|\tilde{w}|}$, then

$$w = \tilde{w}\left(1 - \frac{\lambda S^{-2}}{2|\tilde{w}|}\right).$$

In the case $\text{sign}(w) = \text{sign}(\tilde{w}) = 1$, we have

$$w_i = \underbrace{\tilde{w}_i}_{>0}\left(1 - \frac{\lambda S_i^{-2}}{2|\tilde{w}_i|}\right) > 0,$$

meaning that

$$w_i = \tilde{w}_i \max\left(0, 1 - \frac{\lambda S_i^{-2}}{2|\tilde{w}_i|}\right).$$

In the case $\text{sign}(w) = \text{sign}(\tilde{w}) = -1$, we have

$$w_i = \underbrace{\tilde{w}_i}_{<0}\left(1 - \frac{\lambda S_i^{-2}}{2|\tilde{w}_i|}\right) < 0,$$

meaning that

$$w_i = \tilde{w}_i \max\left(0, 1 - \frac{\lambda S_i^{-2}}{2|\tilde{w}_i|}\right).$$

Combining the two cases leads to the closed form solution that is called ***soft thresholding***,

$$w_i = \tilde{w}_i \max\left(0, 1 - \frac{\lambda S_i^{-2}}{2|\tilde{w}|}\right),$$

where λ removes components that would change the sign of the solution, leading to a sparse solution. Sadly, this analysis is only valid in the eigenspace of X, but the notion of soft thresholding carries on to the general case and is used in several algorithms.

2.3.2 *Ridge/Tikhonov*

The second popular regularization method, probably even more popular than the LASSO, is the Ridge regularization, also known as Tikhonov regularization. It stems from the pseudo-inverse closed form solution of the least-square regression that requires $X^\top X$ to be invertible. The main issue is that if $X^\top X$ has small eigenvalues (***e.g.***, $\text{rank}(\mathbf{X}) < d$), then the inverse will have very large values which will bleed into the solution.

We thus need a regularizer that avoid large values in the solution, which fortunately we can do by penalizing the ℓ_2 norm of w. This leads to solving the ERM problem with the augmented risk

$$\mathcal{L}(w) = \frac{1}{n}\sum_i \ell(\mathrm{x}_i^\top w, y_i) + \lambda\|w\|^2,$$

where λ is a hyper-parameter that sets the trade-off between fitting the data with the loss ℓ and regularizing w with $\|w\|^2$.

Contrarily to the LASSO, we can obtain a closed form solution for least-square regression with Tikhonov regularization. Setting the derivative of the augmented risk to 0, we get

$$\frac{\partial \mathcal{L}}{\partial w} = 0 = \frac{2}{n}(X^\top X w - X^\top Y) + 2\lambda w,$$

which gives us

$$w = (X^\top X + n\lambda I)^{-1} X^\top Y.$$

The effect of the regularization is offsetting all eigenvalues in the covariance matrix by λ, which is where its name ***ridge*** comes from.

Adding entries on the diagonal on the covariance matrix has two effects: First, it makes sure the entire space is spanned by the data making the inversion possible (and numerically stable provided λ is well above machine precision). Second, it compresses the importance of the larger dimensions, making the original data look more and more like a sphere. A similar effect could be obtained by a whitened PCA, which consists in multiplying the PCA projector by the inverse square root of the corresponding eigenvalues such that the covariance matrix in the projected space is the identity.

2.4 Exercises

Exercise 2.1 — Reconstruction without norm constraint. Show that the ERM problem with the reconstruction loss of Equation 2.1 but without the norm constraint collapses to $\|w\| = 1$.

Exercise 2.2 — Geometric interpretation of LASSO and Ridge. Using the geometric picture of a loss's level sets intersecting an ℓ_1 ball vs an ℓ_2 ball, explain why LASSO induces sparsity and ridge does not.

Exercise 2.3 — Regularization of polynomial features. Implement polynomial and periodic feature maps for 1D regression. Fit increasing polynomial degrees on a nonlinear target, then add ridge regularization and show how it controls the under/overfitting trade-off.

3

Kernels and Support Vector Machines

How do we design a non-linear learning algorithm that has its complexity inherently controlled so as to prevent overfitting? This interesting question was raised in the previous chapter where we saw that we can indeed map our input vectors x using a non-linear mapping ϕ, but then the choice of the complexity of ϕ becomes a source of potential overfitting.

In this chapter, we will investigate this question by introducing two new concepts that have been very successful before the advent of deep learning, namely ***Structural Risk Minimization*** for controlling the impact of increased complexity on generalization, and ***kernels*** that generalize the idea of mapping the input with a non-linear mapping and give rise to entire families of natively non-linear methods.

3.1 Structural Risk Minimization

Let us assume we have access to a number $m_{\mathcal{H}}$ that quantifies the complexity of a predictor family $\mathcal{H}$. Given a family $\mathcal{H}$ of predictors that all have an empirical risk $\hat{\mathcal{R}}_{\mathcal{A}} = 0$ on some dataset $\mathcal{A}$ and that can be split into subsets S_k ordered by their complexity $m_{\mathcal{H}}$,

$$S_0 \subset S_1 \subset \cdots \subset S_N,$$

with

$$m_{S_0} \leq m_{S_1} \leq \cdots \leq m_{S_N},$$

It seems intuitive to select a predictor from S_0 so as to minimize the chance that our predictor has extra capacity that would lead to overfitting.

This is a Machine Learning version of Occam's razor: Among all possible solutions, the simplest is the best. Here we assumed that all predictors have the same null empirical risk, and so they are all equivalent and we do not lose performances when selecting the simplest.

Can we actually prove that Occam's razor is a good idea for machine learning? Indeed we actually can derive bounds on the true risk from the complexity of the predictor family. These bounds are a combination of the empirical risk and the risk endured by considering a predictor family of higher complexity, allowing the practitioner to trade one for the other. Optimizing with respect to such bounds is called ***Structural Risk Minimization***, which is a bit of a misnomer since the structural risk is not an approximation of the true risk (whereas the empirical risk is), but an upper bound to it. As such, a high structural risk does not imply a high true risk, but only a low confidence in the actual performances of the selected predictor.

3.1.1 Shattering coefficient

Let us focus on the task of binary classification and take into account the decisions in $\{-1, 1\}$ taken by the predictors of the family $\mathcal{H}$. When sampling a dataset $\mathcal{A} = \{(x_i, y_i)\}_{1 \leq i \leq m}$ of size m, there are exactly 2^m possibilities for the labels y_i. As such, there are only at most 2^m predictors from $\mathcal{H}$ that are *distinct* on $\mathcal{A}$. In fact, there may very well be much less than that. For example, if $\mathcal{H}$ is the family of constant predictors, there are only 2 different predictors, regardless of m.

Let us denote $\hat{m}_{\mathcal{H}}(\mathcal{A})$ the number of predictors from $\mathcal{H}$ that are distinct on $\mathcal{A}$, by which we mean that their outputs are not identical on $\mathcal{A}$. We define the ***shattering coefficient*** of $\mathcal{H}$ as follows:

Definition 3.1 (Shattering coefficient)
The shattering coefficient of a predictor family $\mathcal{H}$ is defined as

$$m_{\mathcal{H}} = \sup_{|\mathcal{A}|=m} \hat{m}_{\mathcal{H}}(\mathcal{A}), \tag{3.1}$$

over all possible datasets $\mathcal{A}$ of size m.

Notice that the shattering coefficient is independent of a specific dataset. It is simply the number of different labelings that can be produced by $\mathcal{H}$ for datasets of size m. If $m_{\mathcal{H}} = 2^m$ then all possible labelings can be produced and the family $\mathcal{H}$ is said to ***shatter*** m points.

It is somewhat intuitive that families with a higher shattering number are more complex, since they can solve more prediction problems for a given dataset size.

Using this concept, we can prove the following theorem:

Theorem 3.2 (Risk Upper Bound using Shattering)
Let $\mathcal{H}$ be a predictor family. Then with probability at least $1 - \delta$ over the choice of a dataset $\mathcal{A}$ of size m, we have for any $h \in \mathcal{H}$

$$\mathcal{R}(h) \leq \hat{\mathcal{R}}_{\mathcal{A}}(h) + \sqrt{\frac{8}{m}\left(\ln \mathbb{E}[\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})] - \ln \frac{\delta}{4}\right)},$$

with $\mathcal{A}_{2m}$ an auxiliary dataset of size $2m$, independent of $\mathcal{A}$, and the expectation taken over its distribution.

The proof follows the same union-bound schema we used for the PAC bounds in the first chapter, but it requires the following lemma due to (Vapnik, 1995) that we will not prove here.

LEMMA 3.3 (Symmetrization)
Let $\hat{\mathcal{R}}_n(h)$ denote the empirical risk of a predictor h on a dataset of size n, and $\hat{\mathcal{R}}_{1:n}(h)$ and $\hat{\mathcal{R}}_{n+1:2n}(h)$ be the empirical risk of h on a dataset of size $2n$ computed respectively on the first n elements and the last n elements. For $n\varepsilon^2 \geq 2$, we have

$$P(\sup_{h\in\mathcal{H}}(R(h) - \hat{\mathcal{R}}_n(h)) > \varepsilon) \leq 2P(\sup_{h\in\mathcal{H}}(\hat{\mathcal{R}}_{1:n}(h) - \hat{\mathcal{R}}_{n+1:2n}(h)) > \varepsilon/2),$$

where the first probability is over i.i.d. datasets of size n, whereas the second probability is over i.i.d datasets of size $2n$ split in halves.

Proof of Theorem 3.2. We follow the scheme proposed in (Schölkopf and Smola, 2002). Let use consider the probability that the largest difference between two empirical risks on datasets of size m differ by $\varepsilon/2$,

$$P(\sup_{h\in\mathcal{H}}(\hat{\mathcal{R}}_{1:m}(h) - \hat{\mathcal{R}}_{m+1:2m}(h)) > \varepsilon/2).$$

We can marginalize over all the possible datasets

$$\int_{\mathcal{X}\times\{-1,1\}^{2m}} P(\sup_{h\in\mathcal{H}}(\hat{\mathcal{R}}_{1:m}(h) - \hat{\mathcal{R}}_{m+1:2m}(h)) > \varepsilon/2 \mid \mathcal{A}_{2m}) dP(\mathcal{A}_{2m}),$$

and the question becomes whether we can bound that inner probability.

Because $\mathcal{H}$ has a finite number of distinct predictors on a dataset $\mathcal{A}_{2m}$ of size $2m$, as encoded by $\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})$, the event corresponding to the $\sup_{h\in\mathcal{H}}$ can be decomposed into the union of events corresponding to distinct predictors $h_1, \cdots, h_{\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})}$. If we can bound the probability of occurrence for each distinct predictor h_i, we can bound the union by the sum.

For a single predictor h_i, the difference of empirical risks depends only on the ordering of the samples, which one was in the first half and which one was in the second half. Because the samples are i.i.d. and the 01-loss is binary, this is exactly the difference between the sum of two sequences of Bernoulli trials $\frac{1}{m}\sum_{i=1}^{m}\zeta_i$ and $\frac{1}{m}\sum_{i=m+1}^{2m}\zeta_i$ which we can bound using Hoeffding's inequality by

$$P\left(\frac{1}{m}\sum_{i=1}^{m}\zeta_i - \frac{1}{m}\sum_{i=m+1}^{2m}\zeta_i > \varepsilon\right) \leq 2e^{-m\varepsilon^2/2}.$$

In our case, since we want $\varepsilon/2$, we get that

$$P(\hat{\mathcal{R}}_{1:m}(h_i) - \hat{\mathcal{R}}_{m+1:2m}(h_i) > \varepsilon/2) \leq 2e^{-m\varepsilon^2/8}.$$

Summing over the h_i and computing the expectation reads

$$\int_{\mathcal{X}\times\{-1,1\}^{2m}} P(\sup_{h\in\mathcal{H}}(\hat{\mathcal{R}}_{1:m}(h) - \hat{\mathcal{R}}_{m+1:2m}(h)) > \varepsilon/2 \mid \mathcal{A}_{2m}) dP(\mathcal{A}_{2m}) \leq 2\mathbb{E}[\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})]e^{-m\varepsilon^2/8}.$$

Plugging Lemma 3.3 with $n = m$ leads to the desired bound

$$P(\sup_{h\in\mathcal{H}}(R(h) - \hat{\mathcal{R}}_m(h)) > \varepsilon) \leq 4\mathbb{E}[\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})]e^{-m\varepsilon^2/8}.$$

Setting the right-hand side to δ and solving for ε gives us

$$4\mathbb{E}[\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})]e^{-m\varepsilon^2/8} = \delta,$$

$$\varepsilon = \sqrt{\frac{8}{m}\left(\ln \mathbb{E}[\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})] - \ln\frac{\delta}{4}\right)},$$

which completes the proof. ■

SRM bounds like the one we demonstrated provide guarantees when the model has low complexity and is trained on many examples. What they do not provide is guarantee of failure when the model has high capacity. We show an illustration of such bounds in Figure 3.1. On the right part of the graph, the true risk is below the bound, but we do not know if it is close to the bound or closer to the empirical risk. What the bound gives us is confidence in the result, which can be expressed as the gap between the empirical risk and the bound. When the gap is the tightest, we are the most confident in the results, but this does not imply that they are good. It could very well be that a point further on the right is much better, but we cannot guarantee it.

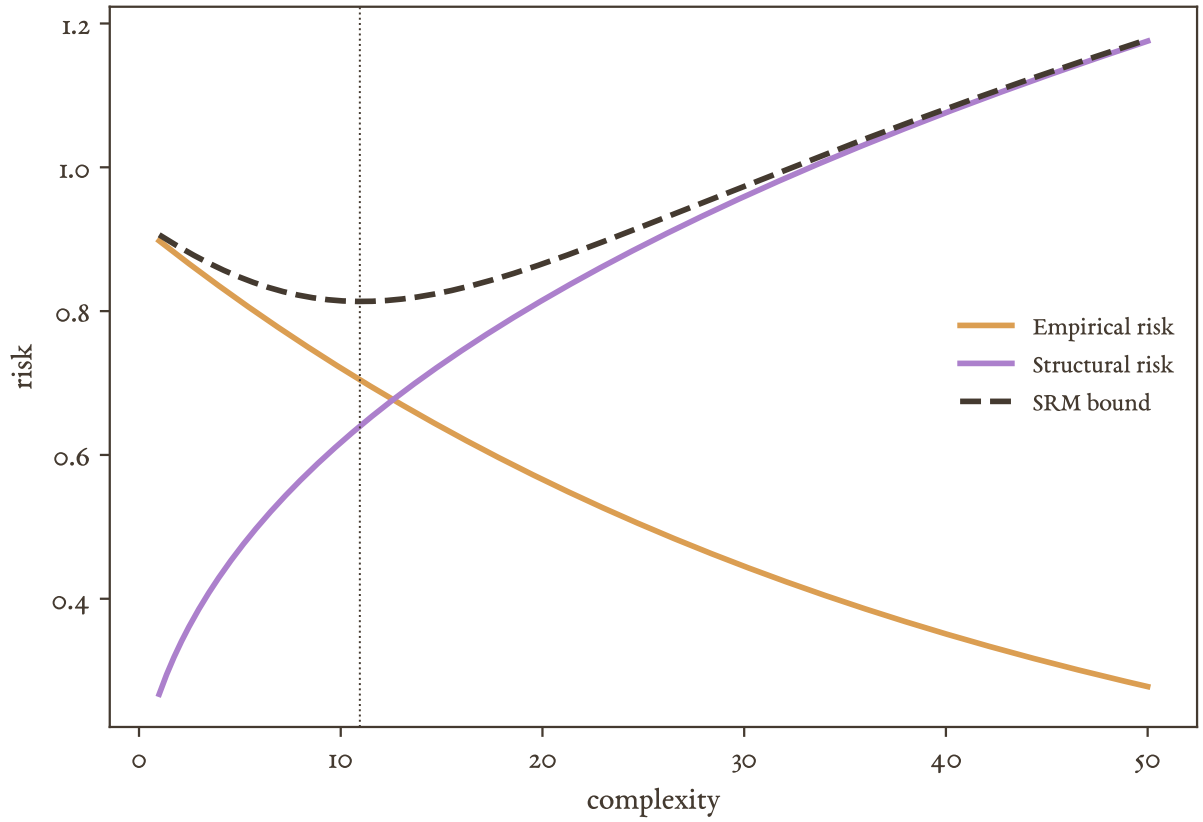

Figure 3.1 Illustration of the structural risk minimization trade-off: the empirical risk decreases with model complexity while the structural (confidence) term of Corollary 3.6 grows with it. The SRM bound (dashed) is a smooth combination of the two, with a minimum (dotted line) at an intermediate complexity.

3.1.2 VC Theory

Theorem 3.2 relies on the expectation of shattering datasets sampled from the data distribution and is thus dependent on the machine learning problem. This makes it very difficult to express in practice as we often cannot characterize this expectation. Instead, we would like an upper-bound that is independent of the problem and characterizes the predictor family ***regardless*** of the data distribution.

It is easy to see that

$$\ln \mathbb{E}[\hat{m}_{\mathcal{H}}(\mathcal{A}_{2m})] \leq \ln m_{\mathcal{H}},$$

a quantity known as the ***growth coefficient*** $\mathcal{G}_{\mathcal{H}}(m)$. If the growth coefficient is dominated by m, then the second term of the bound vanishes and learning is always successful given enough samples. If it is

not dominated by m, which is for example the case when $\mathcal{H}$ has maximal capacity and shatters any dataset $\ln m_{\mathcal{H}} = m \ln 2$, then the second term does not vanish and learning cannot be guaranteed.

The difference between these two cases is encompassed by a key machine learning concept called the ***VC*** dimension, after Vapnik and Chervonenkis (Vapnik, 1995).

Definition 3.4 (VC Dimension)
The VC dimension of a predictor family $\mathcal{H}$ is the maximum number m_{VC} such that

$$G_{\mathcal{H}}(m) = m \ln 2,$$

or infinite if no such number exists. In other words, m_{VC} is the maximum number of points that can be shattered by $\mathcal{H}$.

Proposition 3.5 (Finite classes have finite VC dimension)
If $\mathcal{H}$ is finite, then $m_{VC} \leq \log_2 |\mathcal{H}|$.

Proof. If $\mathcal{H}$ shatters a set $\mathcal{A}$ of size m, then $\hat{m}_{\mathcal{H}}(\mathcal{A}) = 2^m$, so $\mathcal{H}$ contains at least 2^m distinct predictors, giving $|\mathcal{H}| \geq 2^m$. Taking $m = m_{\text{VC}}$ gives the bound. ■

If a predictor family has infinite VC dimension, then learning cannot be guaranteed. On the contrary, if the VC is finite, it can be proven that

$$G_{\mathcal{H}}(m) \leq m_{\text{VC}} \left(\ln \frac{m}{m_{\text{VC}}} + 1 \right),$$

for $m > m_{\text{VC}}$.

We can leverage this fact to provide a more practical corollary to Theorem 3.2.

Corollary 3.6 (Risk Upper Bound using VC dimension)
Let $\mathcal{H}$ be a predictor family and m_{VC} its VC dimension. Then with probability at least $1 - \delta$ over the choice of $\mathcal{A}$ of size m, we have for any $h \in \mathcal{H}$

$$\mathcal{R}(h) \leq \hat{\mathcal{R}}_{\mathcal{A}}(h) + \sqrt{\frac{8}{m} \left(m_{VC} \left(\ln \frac{2m}{m_{VC}} + 1 \right) - \ln \frac{\delta}{4} \right)}.$$

This means that when the number of training points m grows large compared to the VC dimension, the second term of the bound vanishes and learning can be guaranteed.

Finding the VC dimension of a predictor family is not an easy task. One has to prove that a set of size less or equal than the VC dimension can be shattered, in all the 2^m possible labelings, while exhibiting a counter-example for the VC dimension plus one. We provide exercises on the topic at the end of the chapter.

3.2 Support Vector Machines

We now want to put into practice ideas using SRM and the VC theory of generalization. This has led to the development of a successful family of learning algorithms called ***SVM*** for Support Vector Machines.

3.2.1 Linear SVM

Let us momentarily go back to linear predictors. It would be nice if we could leverage the VC theory to get an upper bound on the generalization error, and it turns out that the VC dimension of linear predictors in $\mathbb{R}^d$ is $d + 1$. This already tells us that there are no learning guarantees for datasets of size less than $d + 1$. This is clearly the case for our Bluebell dataset and may explain the strong overfitting we witness in the previous chapter.

Vapnik (1995) showed that we can have a more precise bound and that for a dataset $\mathcal{A} = \{(x_i \in \mathbb{R}^d, y_i)\}$ such that $\|x_i\| \leq R$ and that can be separated by a linear predictor with margin Δ,

$$m_{\text{VC}} \leq \min\left(\frac{R^2}{\Delta^2}, d\right) + 1,$$

where the margin refers to the smallest distance between training samples and their projection on the decision boundary.

For a predictor of the form $h(x) = w^\top x + b$, the decision boundary is the hyperplane orthogonal to w and the margin is inversely proportional to $\|w\|$,

$$\Delta = \min_{x_i \in \mathcal{A}} \frac{|w^\top x_i + b|}{\|w\|}.$$

Following the SRM principle, the key idea of Support Vector Machines, developed by Boser et al. (1992), is to fix that minimum distance to 1, which conveniently is the minimum distance for which the hinge loss is always null. Then, minimizing the VC dimension and thus the bound of Corollary 3.6 corresponds to the following optimization problem on $\mathcal{A}$:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|^2, \\ \text{s.t.} \quad & \forall i, y_i(w^\top x_i + b) \geq 1. \end{aligned}$$

In practice, it is not easy to enumerate predictors that satisfy the hard constraints while minimizing their norm. It could very well be that the problem is not linearly separable and that the solution is an empty set.

To overcome this difficulty, Cortes and Vapnik (1995) defined a soft margin problem that allows us to exchange some structural risk for some constraint violations. This is formalized in the following soft-margin problem:

$$\begin{aligned} \min_{w,b,\zeta_i} \quad & \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \zeta_i, \\ \text{s.t.} \quad & \forall i, y_i(w^\top x_i + b) \geq 1 - \zeta_i, \zeta_i \geq 0. \end{aligned}$$

Instead of directly tackling the constrained problem, we can solve the equivalent problem formulated with the hinge loss,

$$\min_{w,b} \quad \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i(w^\top x_i + b)).$$

This formulation of the problem can be easily solved using SGD, as we show in Listing 3.1, although more involved solvers exist, notably by Bordes et al. (2009), Shalev-Shwartz et al. (2007) and Roux et al. (2012).

Let us analyze Listing 3.1 a bit. The gradient of the hinge loss is 0 when the samples have a margin greater or equal to 1. This is handled in line 21 as a binary mask instead of a conditional statement. On modern architectures like GPU, conditionals are much slower than bitwise operations like comparisons and thus converting conditional paths into masks is always a big win in term of speed. Additionaly, the code presented can only work on datasets of limited sizes because we copy the entire dataset as a tensor at the beginning of the fit method (line 11). This is extremely efficient when executing on GPU because all variables are directly in the GPU memory and thus no slow-memory access cost is endured. On larger datasets, the mini-batch as to be streamed from disk (or worse, the network), and the main bottleneck for the speed would be IO cost since the computations are very light (See **??**).

3.2.2 Dual Problem

Let us consider the hard margin problem once again and compute its Lagrangian $\mathcal{L}(w, b, \alpha)$ on the training set $\mathcal{A} = \{(x_i, y_i)\}$

$$\begin{aligned} \mathcal{L}(w, b, \alpha) \quad &= \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i (y_i (w^\top x_i + b) - 1), \\ \text{s.t.} \quad &\forall i, \alpha_i \geq 0. \end{aligned} \tag{3.2}$$

Using the Karush-Kuhn-Tucker optimality conditions allows us to uncover the following proposition for the dual problem.

> Proposition 3.7 (SVM has support vectors)
> *The solution of Problem 3.2 can be written as*
>
> $$w^\star = \sum_{i=1}^{n} \alpha_i y_i x_i,$$
>
> *Moreover the combination is sparse since*
>
> $$y_i (w^{\star\top} x_i + b) \neq 1 \implies \alpha_i = 0.$$

Proof. Consider the derivative of $\mathcal{L}(w, b, \alpha)$ with respect to w at the solution $w^\star$,

$$\frac{\partial \mathcal{L}}{\partial w}(w^\star) = w^\star - \sum_i \alpha_i y_i x_i,$$

and set it to 0:

$$\begin{aligned} w^\star - \sum_i \alpha_i y_i x_i &= 0 \\ w^\star &= \sum_i \alpha_i y_i x_i. \end{aligned}$$

Furthermore, by the complementary slackness condition at the solution, we have

$$\forall i, \alpha_i (y_i (w^{\star\top} x_i + b) - 1) = 0,$$

which means

$$w^{\star\top} x_i + b \neq y_i \implies \alpha_i = 0.$$

■

Listing 3.1 Linear Support Vector Machine with soft margin (hinge loss and Tikhonov regularization) for multiple classes using pytorch following `scikit-learn` interface.

```
import torch

class LinearSVM:
    def __init__(self, n_classes, lam=1e-4, eta0=1e-3, batch_size=32, steps=10000):
        self.n_classes = n_classes
        self.lam = lam
        self.eta0 = eta0
        self.batch_size = batch_size
        self.steps = steps
    def fit(self, X, y):
        X = torch.as_tensor(X, dtype=torch.float32)
        Y = 2 * torch.nn.functional.one_hot(torch.as_tensor(y, dtype=torch.long),
            self.n_classes).float() - 1
        n, d = X.shape
        self.w = torch.zeros(d, self.n_classes)
        self.b = torch.zeros(self.n_classes)
        for t in range(self.steps):
            eta = self.eta0 / (1 + t) ** 0.5
            idx = torch.randint(0, n, (self.batch_size,))
            Xb, Yb = X[idx], Y[idx]
            margin = (Xb @ self.w + self.b) * Yb
            active = (margin < 1).float()
            gw = self.lam * self.w - (Xb.T @ (Yb * active)) / self.batch_size
            gb = -(Yb * active).sum(dim=0) / self.batch_size
            self.w -= eta * gw
            self.b -= eta * gb
        return self
    def decision_function(self, x):
        return torch.as_tensor(x, dtype=torch.float32) @ self.w + self.b
    def predict(self, x):
        return self.decision_function(x).argmax(dim=1).numpy()
```

This is where the name of the method comes from, as the solution is ***supported*** by a limited set of training samples that are exactly ***on*** the margin. It turns out that this decomposition on a subset of the training set is much more general than SVM and is called ***representer theorems***. We will come back to the concept later in the chapter.

3.3 Kernel SVM

Let us recall the Lagrangian 3.2 and derive the dual problem

$$\max_{\alpha} \inf_{w,b} \mathcal{L}(w, b, \alpha)$$
$$\text{s.t.} \quad \forall i, \alpha_i \geq 0.$$

Since $w^\star = \sum_i \alpha_i y_i x_i$, we can constrain the problem to this decomposition, which leads to removing w from the optimization problem. Similarly, using the optimality condition on b leads to the constraint $\sum_{i=1}^n \alpha_i y_i = 0$ which also makes b disappear from $\mathcal{L}$. We obtain the following expression involving the dual variables α_i only,

$$\inf_{w,b} \mathcal{L}(w, b, \alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^\top x_j.$$

The dual SVM problem is thus simply

$$\max_{\alpha} D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^\top x_j.$$

Notice that $D(\alpha)$ does not depend on x but only on $\langle x_i, x_j \rangle$. As such, we can map x to a (higher VC-dimensional) space using a non-linear mapping $\phi(x)$ to get a predictor of increased complexity without needing to provide an explicit ϕ. Instead, we only need a function $k(\cdot, \cdot)$ such that $k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$, that is called a ***kernel*** and defines the ***similarity*** between x_i and x_j.

We can rewrite the dual SVM problem to make the kernel visible,

$$\max_{\alpha} D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j), \tag{3.3}$$

$$\text{s.t. } \forall i, \alpha_i \geq 0, \sum_{i=1}^n \alpha_i y_i = 0, \tag{3.4}$$

or in matrix form:

$$\max_{\alpha} D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} (\alpha \circ Y)^\top K (\alpha \circ Y),$$
$$\text{s.t.} \alpha \geq 0, \alpha^\top Y = 0,$$

with $\circ$ the Hadamard (element wise) product, and K the Gram matrix of the kernel.

The kernelized version was already in the original article by Boser et al. (1992), although no good software was available at the time to efficiently solve the corresponding quadratic problem.

We can execute the same machinery to obtain the dual problem for the soft-margin SVM. The Lagrangian then takes the form

$$\mathcal{L} = \frac{1}{2} \|w\|^2 + C \sum_i \zeta_i - \sum_i \mu_i \zeta_i - \sum_i \alpha_i (y_i (w^\top x_i + b) - 1 + \zeta_i),$$
$$\text{s.t. } \forall i, \alpha_i \geq 0, \mu_i \geq 0,$$

with $C = 1/(n\lambda)$. It is easy to see that the KKT conditions leads to the same decomposition as in Proposition 3.7. However, examining the derivative with respect to the other primal variable b and dual variables α_i lead to additional constraints, $\sum_{i=1}^{n} \alpha_i y_i = 0$ and $0 \leq \alpha_i \leq C$ respectively.

The soft margin Kernel SVM problem is thus

$$\max_{\alpha} D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j), \tag{3.5}$$

$$\text{s.t.} \quad \forall i, 0 \leq \alpha_i \leq C, \sum_i \alpha_i y_i = 0. \tag{3.6}$$

In both the hard and the soft margin case, the decision function is given by

$$h(x) = \sum_{i=1}^{n} \alpha_i y_i k(x, x_i) + b.$$

The main difference between the hard margin problem and the soft margin one is that support vectors cannot contribute more than C to the solution in the soft margin case. A larger C corresponds to a smaller λ, or less regularization, which is intuitive when looking at the box constraints on α_i.

3.3.1 Kernels

We have seen that the dual SVM problem only involves a kernel function $k(x, x') = \langle \phi(x), \phi(x') \rangle$, without ever requiring to explicit ϕ, a property that is known as the ***kernel trick***. Since the kernel corresponds to an inner product in the space mapped by ϕ, it is symmetric and positive semi-definite ($\forall \alpha_i, \forall \alpha_j, \sum_{ij} \alpha_i \alpha_j k(x_i, x_j) \geq 0$).

Examples of popular kernels include

- Linear: $k(x_i, x_j) = \langle x_i, f x_j \rangle$ which corresponds to the linear SVM;
- Polynomial: homogenous $k(x_i, x_j) = \langle x_i, x_j \rangle^p$ or inhomogenous $k(x_i, x_j) = (1 + \langle x_i, x_j \rangle)^p$ which correspond to the different polynomial mapping presented in chapter 2;
- Gaussian: $k(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}$.

Furthermore, new kernels can be constructed using the following properties:

PROPOSITION 3.8 (Kernel combinations)
Let $k_1(\cdot, \cdot)$ and $k_2(\cdot, \cdot)$ be two kernels and $\alpha \geq 0$ a real value, then

1. $\alpha k_1(\cdot, \cdot)$,
2. $k_1(\cdot, \cdot) + k_2(\cdot, \cdot)$,
3. $k_1(\cdot, \cdot) k_2(\cdot, \cdot)$

are also kernels.

Proof. See ??. ■

Finding the right kernel for a specific problem allows the practitioner to inject ***a priori*** knowledge in the design of the machine learning problem.

3.3.2 *Representer theorems*

Kernels can provide a very solid framework for analyzing ML problems. In particular, if we restrict the family of predictors to a space equipped with a kernel, we can obtain fundamental results on the solutions. Let us begin with the definition of such space.

DEFINITION 3.9 (Reproducing Hilbert Kernel Space)
A Reproducing Kernel Hilbert Space (RKHS) is a Hilbert space $\mathcal{H}$ of functions $h : \mathcal{X} \to \mathbb{R}$ equipped with a kernel $k(\cdot, \cdot)$, for which the pointwise evaluation of $h \in \mathcal{H}$ at x corresponds to the dot product with specific element $\phi_x \in \mathcal{H}$,

$$h(x) = \langle h, \phi_x \rangle,$$

and such that

$$\langle \phi_x, \phi_{x'} \rangle = k(x, x').$$

$k(\cdot, \cdot)$ is said to have the reproducing property.

The theory of RKHS can be a bit overwhelming, especially the original material, but a very good overview can be found in Rakotomamonjy et al. (2005). However, since samples correspond to functions in the RKHS, it is intuitive to see that a function that results from operations involving a set of samples x is limited to combinations of the corresponding basis functions ϕ_x. This is the idea behind the ***representer theorems*** that we reproduce from Schölkopf and Smola (2002).

THEOREM 3.10 (Representer theorem for linear prediction)
Let a training set $\mathcal{A} = \{(x_i, y_i)\}$ of size n, an arbitrary error measuring function $\ell(\cdot, \cdot)$ and a strictly increasing function g, then any minimizer of the empirical risk

$$w^\star = \operatorname{argmin}_w \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \langle w, x_i \rangle) + g(\|w\|),$$

admits a decomposition of the form

$$w^\star = \sum_{i=1}^{n} \alpha_i x_i.$$

Proof. Decompose $w^\star$ into the part that exists in the space spanned by $\mathcal{A}$ and the orthogonal residual,

$$w^\star = w_{\mathcal{A}} + w_\perp = \sum_{i=1}^{n} \alpha_i x_i + w_\perp$$

Combine $\ell(y_i, \langle w_{\mathcal{A}}, x_i \rangle) = \ell(y_i, \langle w^\star, x_i \rangle)$ with $g(\|w_{\mathcal{A}}\|) \leq g(\|w^\star\|)$ to obtain the result. ■

THEOREM 3.11 (Representer theorem for RKHS)
Let a training set $\mathcal{A} = \{(x_i, y_i)\}$, $\mathcal{H}$ a Hilbert space of function associated with the reproducing kernel k, an arbitrary error measuring function $\ell(\cdot, \cdot)$ and a strictly increasing function g, then any minimizer $h \in \mathcal{H}$ of the empirical risk

$$h^\star = \operatorname{argmin}_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, h(x_i)) + g(\|h\|_{\mathcal{H}})$$

admits a decomposition of the form

$$h^\star = \sum_i \alpha_i k(\mathrm{x}_i, \cdot).$$

Proof. See ??. ■

Representer theorems tell us that following the ERM, even with a regularization, the solution cannot escape the training set. It is a fundamental limitation of the ERM principle: because we only use data, we only get what is in the data and nothing more. It is a very powerful statement that limits the creativity of an learning machine to combination of what it has already seen, granted that, even for medium sized datasets, such combination can be gigantic as generative AI has recently empirically demonstrated.

3.3.3 *Solvers*

Now that we know more about kernel SVM, we need to study how to solve the corresponding optimization problem.

SDCA. We show below a stochastic algorithm that solves the dual problem 3.5 due to Shalev-Shwartz and Zhang (2013). In the version shown below, we limit ourselves to $b = 0$ since this removes the constraint on $\sum_{i=1}^n \alpha_i y_i = 0$ and makes the optimization much simpler.

Algorithm 3.1 Stochastic Dual Coordinate Ascent (SDCA)

Input: Training set $\mathcal{A} = \{(x_i, y_i)\}_{1 \le i \le n}$, kernel $k(\cdot, \cdot)$, box constraint C, horizon T
Output: Dual variables α

$\alpha \leftarrow \mathbf{0}$
for $t \leftarrow 0$ **to** $T - 1$ **do**
  Sample i uniformly at random in $\{1, \dots, n\}$
  $z_i \leftarrow y_i \sum_j \alpha_j y_j k(x_i, x_j)$
  $\alpha_i \leftarrow \alpha_i + \dfrac{1 - z_i}{k(x_i, x_i)}$
  $\alpha_i \leftarrow \max(0, \min(C, \alpha_i))$
return α

The key idea is to perform stochastic coordinate ascent with an approximate second order term. The Hessian is simply the kernel matrix K and using a diagonal approximation leads to the division by $k(x_i, x_i)$.

The pytorch code for SDCA is shown in Listing 3.2. We ran the algorithm on the Bluebell dataset to 2000 steps with a inhomogenous polynomial kernel of degree 5 and $C = 1000$ and obtain an average training accuracy of 96.2% and an average validation accuracy of 70.5% over the 5-fold splits. We should mention that for this experiment, the images have been shift to $[-0.5, 0.5]$ and then ℓ_2-normalized, and this preprocessing step has a dramatic impact on the resuts. Skipping the normalization decreases the accuracy by 5 points and dividing by the dimension instead is even worse, getting around 30% accuracy. All thoses steps have to be carefully chosen such that the expressivity of the kernel is in the right middle ground.

Listing 3.2 Kernel SVM training via Stochastic Dual Coordinate Ascent using pytorch following `scikit-learn` interface.

```python
import torch


class SDCA:
    def __init__(self, kernel, C=1.0, steps=10000):
        self.kernel = kernel
        self.C = C
        self.steps = steps
    def fit(self, X, y):
        X = torch.as_tensor(X, dtype=torch.float32)
        y = torch.as_tensor(y, dtype=torch.float32)
        n = X.shape[0]
        K = self.kernel(X, X)
        alpha = torch.zeros(n)
        for t in range(self.steps):
            i = torch.randint(0, n, (1,)).item()
            z = y[i] * (alpha * y * K[i]).sum()
            alpha[i] = (alpha[i] + (1 - z) / K[i, i]).clamp(0, self.C)
        self.X, self.y, self.alpha = X, y, alpha
        return self
    def decision_function(self, x):
        x = torch.as_tensor(x, dtype=torch.float32)
        K = self.kernel(x, self.X)
        return K @ (self.alpha * self.y)
    def predict(self, x):
        return torch.sign(self.decision_function(x)).numpy()
```

This is the best result obtained so far, and it is the one that has the most complex predictor family. We can compute the corresponding VC dimension and it is much larger than the size of the dataset. This result illustrates that the SRM bounds we detailed earlier in the chapter are just bounds: whereas the actual true risk cannot be larger than the bound, it can be much lower. What is happening here is that the problem is naturally complex (classifying images from pixels is complex) and requires an absurdly complex predictor that does not meet the requirement for the VC theory to be predictive. At best, we can see that there is overfitting (the training accuracy is much higher than the validation one), but the model is so much better at solving the problem that its validation accuracy is still better than other simpler models that do not overfit as much.

SMO. The first fast SVM algorithm, Sequential Minimal Optimization (SMO) by Platt (1998) is worth mentioning since it is a beauty of optimization engineering. SMO considers the problem with the bias b and thus is subject to the constraint $\sum_{i=1}^{n} \alpha_i y_i = 0$ and the box constraint on the α_i. As such, we cannot optimize a single α_i independently of the others.

The key observation is that if we wish to increase an α_i, then we have to either decrease another one sharing the same label, or increase another one of opposite label to satisfy the sum constraint. This is where the *minimal* in SMO comes from.

Combining that with the box constraint, we can search pairs of samples that satisfy these conditions and try to optimize them within the box constraints. Once a pair has been found, the optimal α_i, α_j have a closed form solution, which makes the actual update step very fast.

The pseudo-code for the SMO algorithm is given below. The algorithm was designed at a time where a few thousand samples were considered a ***large*** dataset and computer were relatively slow (not so much clock-wise than in terms of operations per second). As such, data access was never the bottleneck, but computation was. On a modern architecture like a GPU, SMO would not use the parallel capabilities very well and be slowed-down by the IO cost of streaming pairs of samples from a slow memory.

Algorithm 3.2 Sequential Minimal Optimization (SMO)

Input: Training set $\mathcal{A} = \{(x_i, y_i)\}_{1 \leq i \leq n}$, kernel $k(\cdot, \cdot)$, box constraint C, horizon T
Output: Dual variables α, bias b

$\alpha \leftarrow \mathbf{o}, b \leftarrow 0$
for $t \leftarrow 0$ **to** $T - 1$ **do**
  Sample i, j distinct, uniformly at random in $\{1, \dots, n\}$
  $E_i \leftarrow \sum_k \alpha_k y_k k(x_i, x_k) + b - y_i \;\; E_j \leftarrow \sum_k \alpha_k y_k k(x_j, x_k) + b - y_j$
  if $y_i \neq y_j$ **then**
    $L \leftarrow \max(0, \alpha_j - \alpha_i), \quad H \leftarrow \min(C, C + \alpha_j - \alpha_i)$
  else
    $L \leftarrow \max(0, \alpha_i + \alpha_j - C), \quad H \leftarrow \min(C, \alpha_i + \alpha_j)$
  $\eta \leftarrow 2k(x_i, x_j) - k(x_i, x_i) - k(x_j, x_j)$
  if $L \neq H$ ***and*** $\eta < 0$ **then**
    $\alpha_i^{\text{old}} \leftarrow \alpha_i, \quad \alpha_j^{\text{old}} \leftarrow \alpha_j$
    $\alpha_j \leftarrow \max\left(L, \min\left(H, \; \alpha_j - y_j(E_i - E_j)/\eta\right)\right)$
    $\alpha_i \leftarrow \alpha_i + y_i y_j (\alpha_j^{\text{old}} - \alpha_j)$
    $b_1 \leftarrow b - E_i - y_i(\alpha_i - \alpha_i^{\text{old}})k(x_i, x_i) - y_j(\alpha_j - \alpha_j^{\text{old}})k(x_i, x_j)$
    $b_2 \leftarrow b - E_j - y_i(\alpha_i - \alpha_i^{\text{old}})k(x_i, x_j) - y_j(\alpha_j - \alpha_j^{\text{old}})k(x_j, x_j)$
    if $0 < \alpha_i < C$ **then**
      $b \leftarrow b_1$
    else
      if $0 < \alpha_j < C$ **then**
        $b \leftarrow b_2$
      else
        $b \leftarrow (b_1 + b_2)/2$
return α, b

3.4 Kernelization

Equipped with various kernels and the representer theorems, we can now try to ***kernelize*** various algorithms, or in other words, see if we can derive non-linear variant using kernels.

3.4.1 Kernel Ridge Regression

Let us recall the Ridge regression ERM problem using a mapping ϕ,

$$\min_w \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}(y_i - \langle w, \phi(x_i)\rangle)^2,$$

that has the pseudo-inverse solution

$$w = (\phi(X)\phi(X)^\top + \lambda I)^{-1}\phi(X)Y,$$

with ϕ applied to the columns of X corresponding the samples.

We can leverage the representer theorem to express w as a combination of the $\phi(x_i)$,

$$w = \sum_i \alpha_i \phi(x_i).$$

Then, using the following identity (Woodbury),

$$\left(P^{-1} + B^\top R^{-1} B\right)^{-1} B^\top R^{-1} = PB^\top \left(BPB^\top + R\right)^{-1}$$

with $P^{-1} = \lambda I, R = I, B = \phi(X)$, we get

$$w = \phi(X)(K + \lambda I)^{-1} Y.$$

Thus, we can solve for α and get

$$\alpha = (K + \lambda I)^{-1} Y.$$

Notice how the ridge is now on the kernel matrix. Similar ideas can be used to kernelize linear algorithms, like PCA (see **??**).

3.4.2 Random features

The main downside of kernel methods is their complexity. Most algorithms are much more efficient if you can compute and store the kernel matrix in memory, because otherwise, evaluating the kernel becomes a computational bottleneck. However, the kernel matrix consumes $O(n^2)$ in memory, which quickly becomes intractable even for small datasets.

Instead, if we can find an explicit mapping that approximates the kernel,

$$k(x_i, x_j) \approx \langle \phi(x_i), \phi(x_j)\rangle,$$

we can map the training set,

$$\forall i, \bar{\mathbf{x}_i} = \phi(\mathbf{x_i}),$$

and be back to the complexity of linear methods. This is a different interpretation of the generalized linear models that relies on kernel approximation.

The simplest of such approximation is called the Nyström approximation and was proposed for machine learning by Williams and Seeger (2000). It consists in performing a low-rank approximation of the kernel matrix

$$K = ULU^\top,$$

And keeping only the m leading eigenvectors. The non-linear projection

$$\phi(x) = L_m^{-1/2} U_m^\top K(x), \quad K(x) = [k(x_i, x)]_{1 \leq i \leq n}$$

approximates the mapping of the kernel and limits the VC dimension to m.

Alternatively, a shift-invariant kernel, ***i.e.*** $k(x, x') = k(x - x')$, like the Gaussian kernel is more difficult to approximate. A randomized approximation was proposed by Rahimi and Recht (2007) and consists in random projections combined with sin and cos functions. More precisely, they consider the mapping

$$\begin{aligned} \phi : &\mathbb{R}^d \mapsto \mathbb{R}^{2D} \\ &x \to \frac{1}{\sqrt{D}} [\cos(\omega_1^\top x), \sin(\omega_1^\top x), \dots, \cos(\omega_D^\top x), \sin(\omega_D^\top x)], \end{aligned}$$

with $\omega_1, \dots, \omega_D$ random vectors sampled from the normal distribution. Using the trigonometric identity $\cos(a-b) = \cos(a)\cos(b) + \sin(a)\sin(b)$, we can see that $\langle \phi(x), \phi(x') \rangle = \frac{1}{D} \sum_{k=1}^{D} \cos(\omega_k^\top (x - x'))$, which concludes a question raised in chapter 2 when we studied generalized linear models for periodic signals. More importantly, the authors show that this approximation is an unbiased estimator of the Gaussian kernels and converges to it in probability as $D \to \infty$.

3.5 Gaussian Processes

Now that we are accustomed to kernels, we can consider a powerful learning algorithm, albeit difficult to conceptualize, namely Gaussian processes, or GP for short. Let us consider a predictor h for which the evaluation at a set of points $X = \{x_1, \dots, x_n\}$ is given by

$$h(X) \sim \mathcal{N}(m(X), k(X, X)),$$

with $m(X)$ a function that provides the mean and $k(X, X)$ a function that provides the covariance of the normal distribution we sample from. In other words, the evaluation of h at X is a Gaussian variable with a specific mean and covariance that depends on the other points that are evaluated.

One thing needs to be highlighted here: $k(X, X)$ needs to be positive definite as it is a covariance matrix, so kernels are well suited for that task, and it models how the samples in X are related to one another. A popular choice is to use the Gaussian kernel, but other kernels that encode specific properties of the problem (such as periodicity) can be used.

In practice, we have a training set of known samples X_1 associated with the labels Y_1, and we want to evaluate h at another set of points X_2 leading to the prediction $\hat{Y}_2$. X_1 are called ***anchor points*** or ***collocation points***.

We assume that Y_1 and $\hat{Y}_2$ come from the same multivariate Gaussian:

$$\begin{bmatrix} Y_1 \\ \hat{Y}_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right),$$

with $\Sigma_{ij} = k(X_i, X_j)$. In practice, Y_1 is noisy, and thus $\Sigma_{11} = k(X_1, X_1) + \sigma^2 I$, with σ modeling the noise variance.

The conditional probability is given by

$$P(\hat{Y}_2 | Y_1, X_1, X_2) = \mathcal{N}(\mu_{2|1}, \Sigma_{2|1})$$

with

$$\mu_{2|1} = \mu_2 + \Sigma_{21}\Sigma_{11}^{-1}(Y_1 - \mu_1) = \Sigma_{21}\Sigma_{11}^{-1}(Y_1 - \mu_1),$$

assuming $\mu_2 = 0$, and

$$\Sigma_{2|1} = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}.$$

The cost of training a Gaussian process on the collocation points is mostly in inverting the kernel matrix of the training set. This has a computational cost of $O(n^3)$ without specific properties of the kernel, and a memory cost of $O(n^2)$, which makes them very costly for larger datasets. Nonetheless, they naturally provide the uncertainty associated with the prediction because it is by design a Gaussian distribution.

We show in Figure 3.2 the results of a GP on the same regression problem that we illustrated with polynomial mappings in chapter 2. With a well chosen kernel, GP can perform very well while providing confidence intervals.

3.6 Exercises

Exercise 3.1 — VC dimension of linear classifiers. Show that the VC dimension of linear classifiers in 2D is 3.

Exercise 3.2 — VC dimension of axis aligned rectangles. What is the VC dimension of axis-aligned rectangle?

Exercise 3.3 — Representer theorem for KRR. Prove the representer theorem for kernel ridge regression.

Exercise 3.4 — Random Fourier Features approximation error. Implement random Fourier features approximating an RBF kernel; show empirically that the approximation error shrinks as the number of features grows.

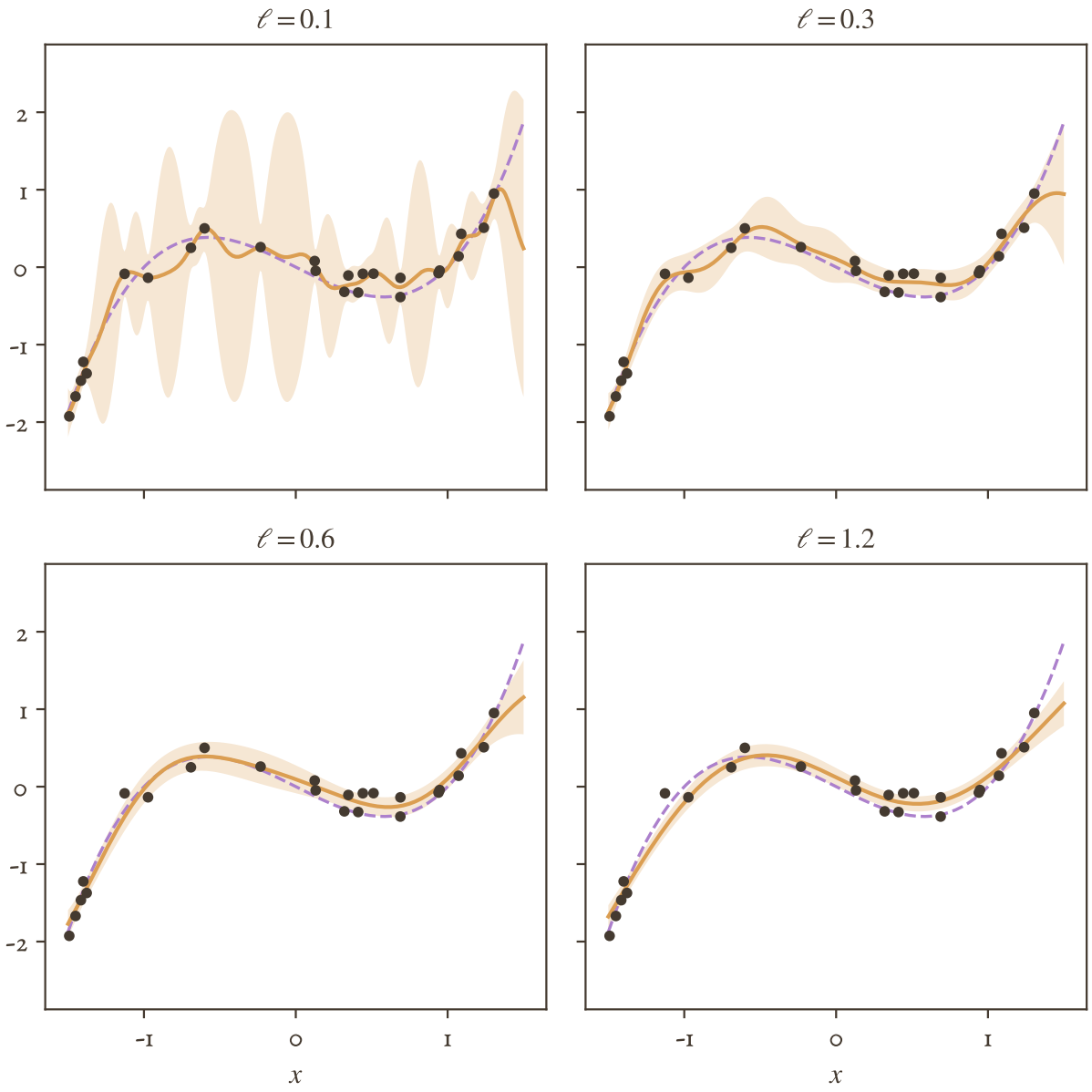

Figure 3.2 Gaussian process regression on the same noisy cubic dataset as Figure 2.6, for varying lengthscale ℓ of the Gaussian kernel. Solid line: posterior mean; shaded area: 95% confidence interval ($\pm 1.96\sigma$); dashed line: true curve; dots: training samples.

4

Decision Trees

Let us consider a finite space $\mathcal{X}_n = \{x_1, \dots, x_n\}$ of size n and consider the set of predictors $\mathcal{H}_n = \{h : \mathcal{X}_n \mapsto \{-1, 1\}\}$ for the binary classification task. Essentially, a predictor from $\mathcal{H}_n$ is a list of mappings between x_i and $\{-1, 1\}$ and can be quite large depending on n.

Is there way to represent h that is more compact than enumerating all the associations? Indeed, since there are only n of such associations, $1 + \log_2 n$ bits are sufficient to encode them each (the $+1$ is for encoding the class). This binary encoding of h corresponds to a partitioning of $\mathcal{X}_n$ using a binary tree and predicting using h corresponds to walking down the tree until a leaf is reached. The value at the leaf corresponds to the prediction.

Using trees as predictors has the advantage of speed as traversal is logarithmic with respect to the number of leaves. In this chapter we investigate tree-based strategies that have been formalized by Breiman et al. (1984) and are still popular in recent times, especially for non-numerical data.

4.1 Decision Regions

Let us consider the ML problem of regressing a scalar value on a vector space $\mathcal{X} = \mathbb{R}^d$ and assume a training set $\mathcal{A}_n$ of size n.

We focus on hierarchical binary partitions of $\mathcal{X}$ that can readily be encoded with a tree so as to benefit from the fast traversal mentioned earlier. A node in such a tree corresponds to a threshold on a specific dimension that splits samples achieving the node into 2 subsets according to their value on that dimension compared to the threshold. An illustration of the partition of $\mathbb{R}^2$ by such a tree is given on Figure 4.1.

For convenience of notations, we will conflate the leaf ℓ_k of the tree and the region of the space it

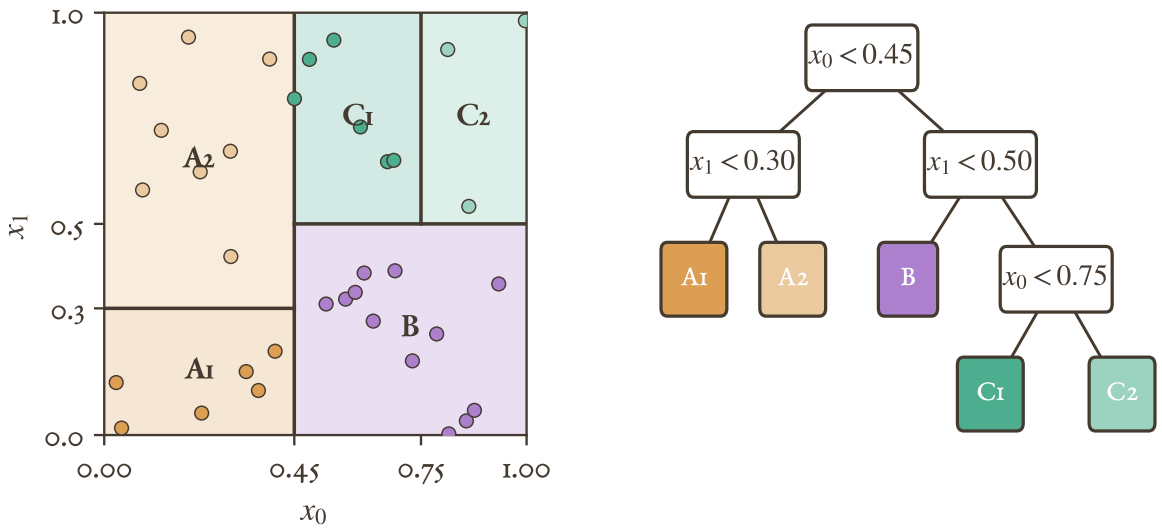

Figure 4.1 Partition of $\mathbb{R}^2$ induced by a binary tree of depth 3 (left) and the corresponding tree, with the splitting dimension and threshold shown at each internal node (right). Leaves are colored to match their region.

encodes. Each leaf is associated with a predicted value and the empirical risk if given by

$$\hat{\mathcal{R}}_{\ell_k}(h) = \frac{1}{|\ell|} \sum_{(x_i, y_i) \in \ell_k} \ell(y_i, h(x_i)),$$

which for the least square problem is minimized by averaging the training samples in ℓ_k.

For a tree of depth L, there are 2^L leaves, and the number of different trees that partition n training samples is

$$2^L! \left\{ {n \atop 2^L} \right\}$$

obtained using a combinatorial argument with the Stirling number of the second kind. The total number of trees is of course infinite, but many such trees produce exactly the same partition and as such the same empirical risk. This number is huge. As an example, $L = 3$ with $n = 10$ leads to over 30 million possible trees, and $L = 4$ with $n = 20$ is over 10^{20}. Sadly, the problem of finding the tree with the lower empirical risk is NP hard and we have to resort to approximate search instead.

The common strategy for training a decision tree is to start for $\mathcal{A}$ and recursively split it among dimensions until the target depth is attained. Starting from a node $\mathcal{N}$, there are only $d \times (|\mathcal{N}| - 1)$ splitting decisions possible. The corresponding thresholds are obtained by ordering the samples along the dimension and taking $|\mathcal{N}| - 1$ values, one between every pair of consecutive samples.

A split is interesting if and only if it decreases the empirical risk and finding the optimal split consists in evaluating the gain in empirical risk obtained by each of the $d \times (|\mathcal{N}| - 1)$ splits and taking the best possible. We define the gain $G(\mathcal{N}, k, \theta)$ of a split on dimension k with threshold θ as the difference between the empirical risk on the original node $\mathcal{N}$ and the sum of empirical risk on the resulting nodes $\mathcal{N}_L$ and $\mathcal{N}_R$, normalized by their population,

> **Definition 4.1** (Gain for Decision Tree growing)
> The gain $G(\mathcal{N}, k, \theta)$ obtained by splitting node $\mathcal{N}$ on dimension k at threshold θ is
>
> $$G(\mathcal{N}, k, \theta) = \hat{\mathcal{R}}_{\mathcal{N}} - \left[\frac{|\mathcal{N}_L|}{|\mathcal{N}|} \hat{\mathcal{R}}_{\mathcal{N}_L} + \frac{|\mathcal{N}_R|}{|\mathcal{N}|} \hat{\mathcal{R}}_{\mathcal{N}_R} \right], \tag{4.1}$$

with μ, μ_L and μ_R the averages of $\mathcal{N}$, $\mathcal{N}_L$ and $\mathcal{N}_R$ respectively.

Using the gain, we can build the following growing strategy that recursively calls itself to grow each branch of the tree. The full decision tree is obtained by running the procedure on $\mathcal{A}$.

Algorithm 4.1 Decision Tree

Input: Node samples $\mathcal{N} = \{(x_i, y_i)\}$, current depth ℓ, maximum depth L
Output: A (sub)tree rooted at $\mathcal{N}$

if $|\mathcal{N}| \leq 1$ *or* $\ell = L$ **then**
  return *Leaf*$(\mu(\mathcal{N}))$
$g^\star \leftarrow -\infty$
for $k \leftarrow 1$ **to** d **do**
  for θ *in candidate thresholds of* $\mathcal{N}$ *along dimension* k **do**
    $g \leftarrow G(\mathcal{N}, k, \theta)$
    if $g > g^\star$ **then**
      $g^\star, k^\star, \theta^\star \leftarrow g, k, \theta$
$\mathcal{N}_L \leftarrow \{(x_i, y_i) \in \mathcal{N} \mid x_i[k^\star] < \theta^\star\}$
$\mathcal{N}_R \leftarrow \{(x_i, y_i) \in \mathcal{N} \mid x_i[k^\star] \geq \theta^\star\}$
return *Node*$\big(k^\star, \theta^\star,$ *GrowTree*$(\mathcal{N}_L, \ell + 1, L),$ *GrowTree*$(\mathcal{N}_R, \ell + 1, L)\big)$

This procedure makes at most $O(2^L)$ calls to itself in order to build all the leaves, and each call has a complexity of $O(nd/2^\ell)$ to compute the optimal decision locally at depth ℓ, thus the complete tree growing is bounded by $O(Lnd)$ which is reasonable for controlled depth. As a comparison, a linear method trained with SGD has a complexity in $O(ndE)$ with E the number of epochs (*i.e.*, number of passes over the entire dataset), but the VC dimension of a tree scales with L, whereas that of the linear model is fixed at $d + 1$. Notice finally that the search for the best dimension and threshold can be performed in parallel for all nodes at a same depth which, assuming sufficient compute to do so, would reduce the runtime complexity to $O(2nd)$.

4.2 Splitting Criteria

So far, we considered the regression case where the optimal leaf prediction is the average of the training samples in the corresponding region. For a classification problem, the predicted value has to be from a finite set, *e.g.* $\{-1, 1\}$ for binary classification. In that case, the empirical risk on a leaf ℓ is obtained using the 01-loss $\ell(\cdot, \cdot)$,

$$\hat{\mathcal{R}}_\ell(h) = \frac{1}{|\ell|} \sum_{(x_i, y_i) \in \ell} \ell(y_i, h(x_i)).$$

We have seen in chapter 1 that minimizing $\hat{\mathcal{R}}_\ell(h)$ is obtained by predicting the label with the highest number of occurrences in ℓ,

$$\forall x \in \ell, h(x) = \arg\max_j \sum_{(x_i, y_i) \in \ell} \mathbb{1}_{y_i = j},$$

with $\mathbb{1}$ the indicator function that outputs 1 if $j = y_i$ and 0 else.

In the case of multiple classes, there can be many splits with the same gain according to the empirical risk, and so the selected one depends on the order of the search. Are they really equivalent?

Consider a case with 4 classes and compare a split that generates proportions [0.0, 0.2, 0.2, 0.6] and [0.6, 0.2, 0.2, 0.0] to another split of the same node that generates proportions [0.0, 0.0, 0.4, 0.6] and [0.6, 0.4, 0.0, 0.0]. They both do the same error (40% wrong prediction on all children), but we can clearly see that the second split leads to nodes that are much easier to split than the first. In fact, the nodes generated by that second split only needs to be split once to attain 0 empirical risk, whereas that is not possible for the first split.

This observation comes from the fact that the empirical risk using the 01-loss is constant except when the dominant class proportion changes. Instead, we would like a loss that is not flat and generates useful signal even when the dominant class varies.

4.2.1 Entropy

Let us denote $p_k(\mathcal{N}) = \sum_{(x_i,y_i)\in\mathcal{N}} \mathbb{1}_{y_i=k}/|\mathcal{N}|$ the proportion of class k in node $\mathcal{N}$. We can make the following information theoretic argument on the complexity of splitting $\mathcal{N}$: describing the content of $\mathcal{N}$ requires an average number of bits that depends on the entropy of $[p_1(\mathcal{N}), \dots]$, and splits are a way to encode that content in a binary string.

As such, a node that has very low entropy would on average require a much lower number of splits than a node with a high entropy. The corresponding cost function to plug in the gain formula 4.1 is

$$C_{\text{ent}}(\mathcal{N}) = -\sum_k p_k(\mathcal{N}) \log p_k(\mathcal{N})$$

Using this criterion, we get 1.37 bits for the first split and 0.97 bits for the second split of the example we gave as motivation at the beginning of this section, thus selecting the second one that was deemed better.

For classification purposes, we are not necessarily interested in a good error rate at any depth, but rather by having a good error rate at the leaves. Minimizing the entropy is thus more a way to minimize the depth required to achieve a good error rate than a strong guarantee on it.

4.2.2 Gini

Suppose that instead of always predicting the class corresponding to the highest p_k, we sample a label from that distribution instead. The expectation of empirical risk taken over the predictions is then

$$\begin{aligned}\mathbb{E}[\hat{\mathcal{R}}_{\mathcal{N}}(h)] &= \frac{1}{|\mathcal{N}|}\sum_{(x,y)\in\mathcal{N}}\sum_k p_k \ell(k,y) = \frac{1}{|\mathcal{N}|}\sum_{(x,y)\in\mathcal{N}}\sum_k p_k \mathbb{1}_{k\neq y} \\ &= \frac{1}{|\mathcal{N}|}\sum_k p_k \sum_{(x,y)\in\mathcal{N}} \mathbb{1}_{k\neq y} = \sum_k p_k(1-p_k).\end{aligned}$$

This quantity is called the ***Gini*** impurity index, and the corresponding cost to plug in the gain definition 4.1 is

$$C(\mathcal{N}) = \sum_k p_k(\mathcal{N})(1-p_k(\mathcal{N})).$$

It is strictly concave (it is in fact a parabola) which makes it also less sensitive to the ordering chosen in the search procedure. Using the Gini impurity index, we obtain a cost of 0.56 for the first split and 0.48 for the second split of the example at the beginning of the section, thus selecting the better split.

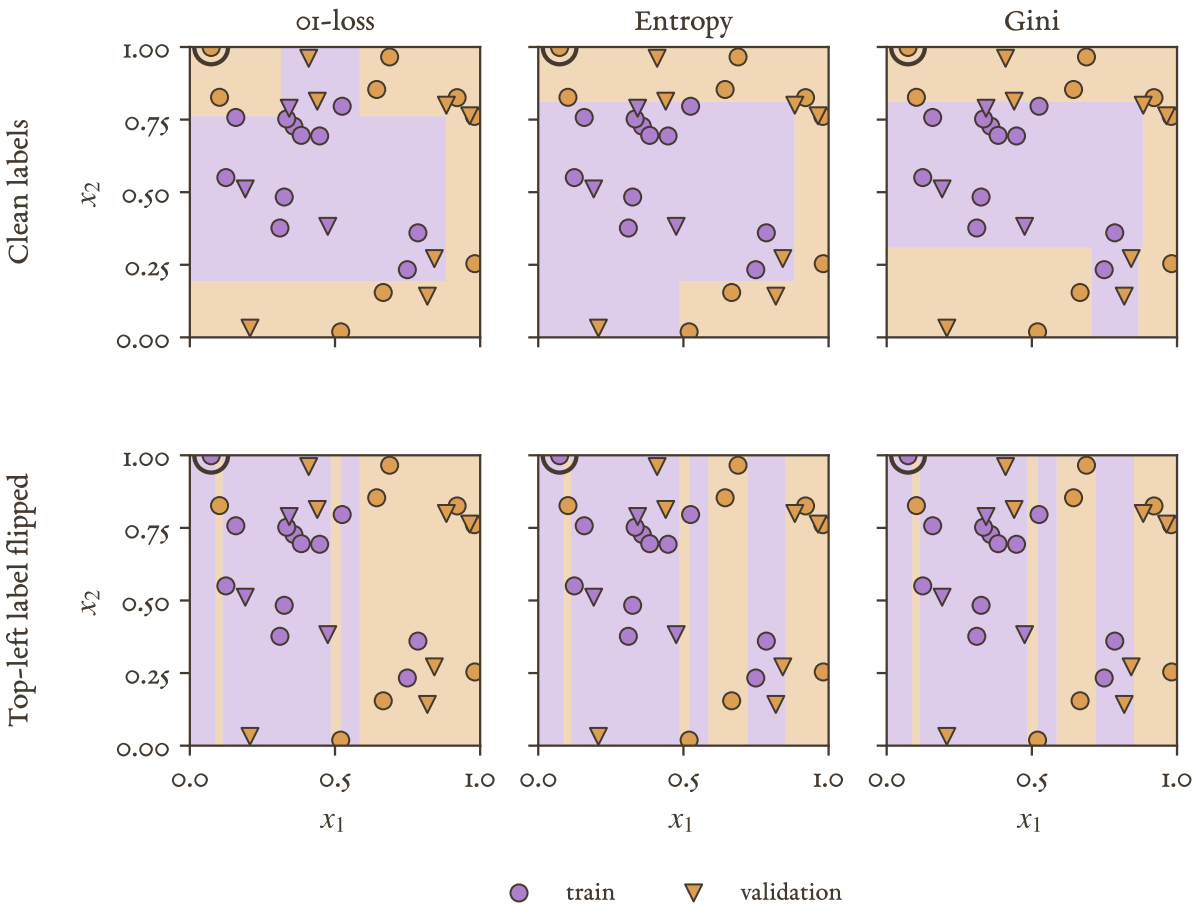

Figure 4.2 *(Top)* Decision boundaries on the synthetic 2D classification set for a depth-3 tree under the three splitting criteria from this section. *(Bottom)* We modify the sample circled in black and flip its label to see the impact of noise label on the tree.

Using Listing 4.1, we trained 3 different trees of depth 5 on the 2 dimensional dataset that was used for the k-NN algorithm in chapter 1. The resulting decision boundary for all three gain measures presented in this chapter are shown in Figure 4.2. The boundaries are different despite all predictors achieving zero empirical error.

Notice however that introducing noise in the label completely changes the decision boundary. This is because doing so can change an early split which has cascading effect on the whole sub-tree.

4.3 Generalization

Because a tree of depth L has at most 2^L leaves, its VC dimension is 2^L as a higher number of points would result in configuration where two points sharing the same leaf can have different labels. This also implies the following proposition if we do not limit the depth of the tree.

Proposition 4.2 (Unbounded trees VC dimension)
Unbounded decision trees have infinite VC dimension.

We have seen that infinite VC dimension is worrying to guarantee that the learning process is working. This raises the question of the consistency of decision tree, that is whether the empirical risk converges to the true risk as the number of samples grows to infinity. This result is difficult to obtain since changing the number of samples changes the boundaries of the leaves and also their associated decision. Instead, using two independent datasets for the thresholds and for the predicted value, as proposed by Athey and Imbens (2016) leads to consistency.

However, it is not realistic to let the depth of the tree grow indefinitely, mainly for practical reasons. In that case, can we guarantee PAC bounds? It turns out that yes, in the form of the following theorem that uses an argument of Minimum Description Length that quantifies the risk of finite families by the complexity in describing them.

Theorem 4.3 (PAC Bound for Decision Trees)
For a tree h of n nodes in dimension d trained on a dataset $\mathcal{A}$ of m samples, we have with probability $1-\delta$

$$\mathcal{R} \leq \hat{\mathcal{R}}_{\mathcal{A}} + \sqrt{\frac{(n+1)\log_2(d+3) + \log_2(2/\delta)}{2m}}.$$

The proof can be found in (Shalev-Shwartz and Ben-David, 2014).

4.4 Implementation

We provide a pytorch implementation of a simple decision tree in Listing 4.1. It accepts all three gain measures discussed in the chapter.

There are several implementation choices that are worth mentioning. First, we make no distinction between a regular node and a leaf and differentiate them only on the returned value.

Second, we perform part of the search in parallel. Namely, for a fixed dimension d, we compute all the gains in parallel (function `_best_split_along` line 32). This is possible by computing cumulative sums of the labels for each of the ordered thresholds which correspond to the predictions of the corresponding split.

We choose to not perform the search in parallel over the dimensions. It could be done by launching

one thread per dimension executing `_best_split_along`. We do not do this in our implementation for simplicity of presentation, but also to be able to test on the Bluebell dataset that has over 12k dimensions which would be too many threads and would thrash the system. An alternative would be to vectorize the search over the dimension in addition to the thresholds, which can be done using the same strategy.

Finally, when growing the tree, we call the `_grow` function (line 69 and 70) with masked versions of the dataset. Depending on the library used, doing so may or may not copy the data and then make the entire speed collapse from data copy.

Listing 4.1 Decision tree classifier: the gain over all candidate thresholds along one dimension is computed in parallel via a cumulative sum; dimensions are searched sequentially.

```
import torch

class Node:
    def __init__(self, dim, threshold, left, right, value=-1):
        self.dim = dim
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

class DecisionTree:
    def __init__(self, max_depth=3, criterion="gini"):
        self.max_depth = max_depth
        self.criterion = criterion
    def fit(self, X, y):
        X = torch.as_tensor(X, dtype=torch.float32)
        y = torch.as_tensor(y, dtype=torch.long)
        self.n_classes = int(y.max()) + 1
        Y = torch.nn.functional.one_hot(y, self.n_classes).float()
        self.root = self._grow(X, Y, depth=0)
    def _cost(self, counts):
        # counts: (..., n_classes) class counts, one row per candidate split.
        p = counts / counts.sum(dim=-1, keepdim=True)
        if self.criterion == "01loss":
            return 1 - p.max(dim=-1).values
        if self.criterion == "entropy":
            logp = torch.where(p > 0, p.log(), torch.zeros_like(p))
            return -(p * logp).sum(dim=-1)
        if self.criterion == "gini":
            return 1 - (p ** 2).sum(dim=-1)
        raise ValueError(f"unknown criterion: {self.criterion}")
    def _best_split_along(self, x, Y):
        n = x.shape[0]
        order = torch.argsort(x)
        x_sorted, Y_sorted = x[order], Y[order]
```

```

        cum_left = torch.cumsum(Y_sorted, dim=0)  # (n, n_classes)
        total = cum_left[-1]
        cum_right = total - cum_left

        left_counts = cum_left[:-1]   # split after the i-th sample, i = 1..n-1
        right_counts = cum_right[:-1]
        n_left = torch.arange(1, n, dtype=x.dtype, device=x.device)
        n_right = n - n_left

        gain = self._cost(total.unsqueeze(0)).squeeze(0) - (
            n_left / n * self._cost(left_counts) + n_right / n *
                self._cost(right_counts)
        )
        i = torch.argmax(gain)
        if not torch.isfinite(gain[i]):
            return float("-inf"), None
        threshold = ((x_sorted[i] + x_sorted[i + 1]) / 2).item()
        return gain[i].item(), threshold
    def _grow(self, X, Y, depth):
        if X.shape[0] <= 1 or depth == self.max_depth:
            return Node(0, 0, None, None, int(Y.sum(dim=0).argmax()))

        best_gain, best_dim, best_threshold = float("-inf"), None, None
        for k in range(X.shape[1]):  # dimensions are searched sequentially
            gain, threshold = self._best_split_along(X[:, k], Y)
            if gain > best_gain:
                best_gain, best_dim, best_threshold = gain, k, threshold
        if best_dim is None:  # every remaining sample is identical
            return Node(0, 0, None, None, int(Y.sum(dim=0).argmax()))

        left = X[:, best_dim] < best_threshold
        return Node(
            best_dim, best_threshold,
            self._grow(X[left], Y[left], depth + 1),
            self._grow(X[~left], Y[~left], depth + 1),
        )
    def predict(self, x):
        x = torch.as_tensor(x, dtype=torch.float32)
        return torch.tensor([self._predict_one(xi) for xi in x])
    def _predict_one(self, xi):
        node = self.root
        while node.value == -1:
            node = node.left if xi[node.dim] < node.threshold else node.right
        return node.value
```

Running this code on the Bluebell dataset with a depth of 16 leads to an average training accuracy

of 97.6% and an average validation accuracy of only 38.5%. This is not a good result and the worst overfitting we have seen so far, but we can understand why by looking at the decisions taken for each class. For each node, we can color the pixels that correspond to the dimensions with a brightness that depends on the proportion of leaves in the right sub-tree (encoding pixel values higher than the threshold) predicting the class, which should indicate that a bright color at that location is characteristic of the class. This produces a template image of the class.

4.4.1 Interpretability

We show the template obtained in Figure 4.3. They are mostly black because the tree does not use many pixels, even at a depth of 16, mostly because the dataset size is much smaller than the dimension of the images.

Interpretability is one of the main appeals of Decision Tree. Since the decision consists in a sequence of binary steps, usually short because the depth is constrained, it is human-readable. Another aspect of interest is the ability to seamlessly combine categorical and numeric data. Dimensions do not have to be of the same nature because they are never compared one against another. Thus, the first dimension could be for example a numerical value like the amount of sunlight received, and the second dimension a categorical one like for example the month. A decision tree handles this discrepancy perfectly fine, it gets a real valued threshold for the first dimension and a boolean expression for the second one.

4.4.2 Stopping rules

In our implementation, we stop growing the tree once a specific depth is reached. While this bounds the complexity of the tree generation process, it is also very limiting as it encourages balanced trees. Instead, one could stop once a threshold on the size of $\mathcal{N}$ is reached. This also limits overfitting by avoiding putting each training sample in a dedicated leaf.

4.5 Exercises

Exercise 4.1 — Gini and entropy comparison. Show that Gini impurity is a quadratic approximation to entropy, and compare their behavior near $p = 0$ vs $p = 0.5$.

Exercise 4.2 — Stopping rules and overfitting. Implement stopping rules (max depth, min samples per leaf) and plot train/validation error as a function of tree depth, then identify where overfitting begins.

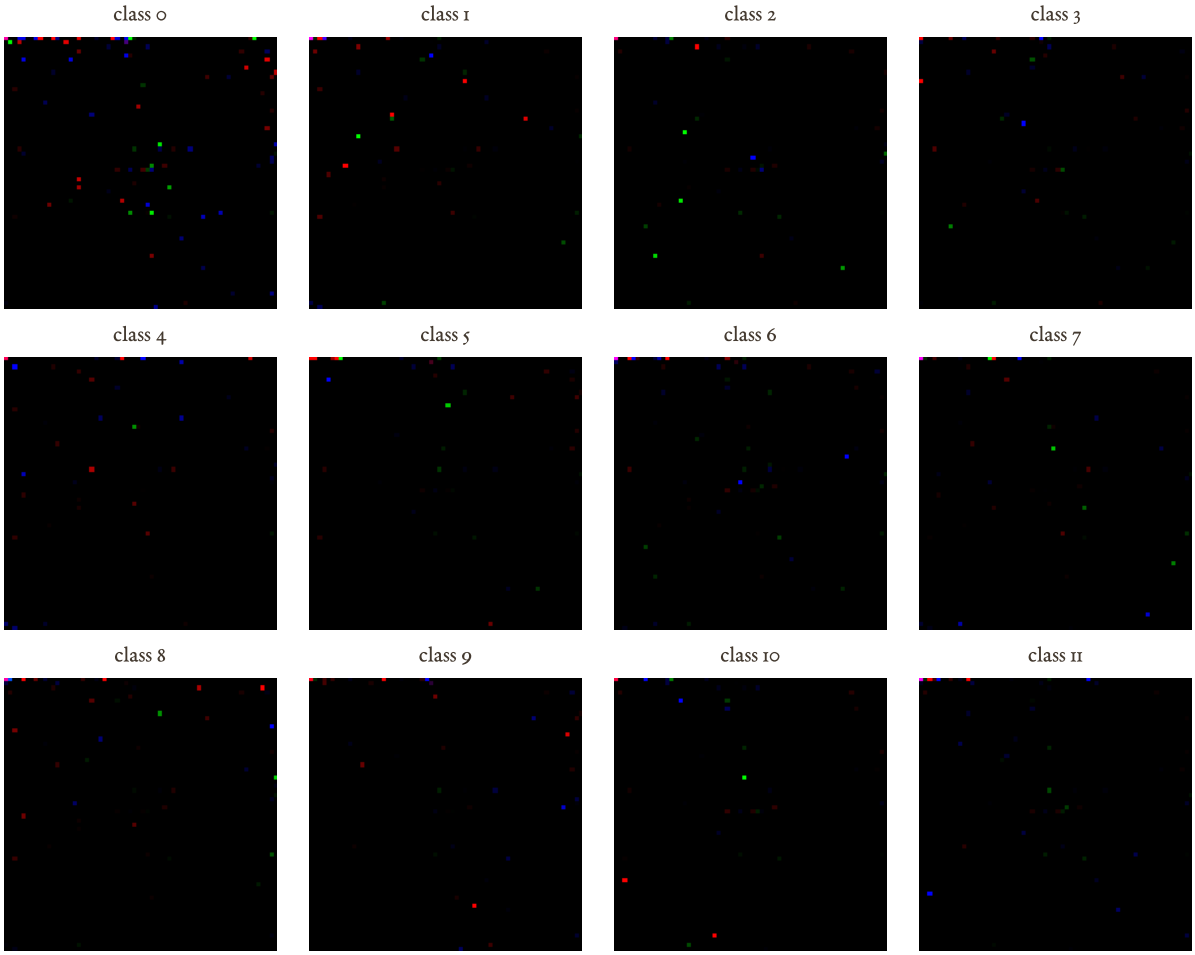

Figure 4.3 Template image derived from the first tree's decision thresholds on the first cross-validation split, for each of the 12 classes.

5

Ensembling

So far we have seen predictors that do not perform extremely well on our small Bluebell dataset. All of them are subject to overfitting in one way or another, even the ones reaching the best performances like kernel SVM.

Is it because the proposed predictor families are not effective for that task? Is it because the algorithms following the ERM and the SRM principles are not able to find a good predictor within the family? How do we correct the shortcomings of one or the other? This chapter explores answers to these questions by leveraging the idea of training several predictors for the same task.

5.1 Bias/Variance trade-off

Let us consider a predictor $h_\theta \in \mathcal{H}$ parametrized by θ sampled from the distribution of parameters output by the learning algorithm, on a sample (x, y), and let us consider the best possible predictor h such that $y = h(x) + \varepsilon$ with $\mathbb{E}[\varepsilon] = 0$ and $\text{Var}[\varepsilon] = \sigma^2$. The least-square error at x can be expressed in relation to the best possible predictor as

$$\begin{aligned}
\mathbb{E}[(h_\theta(x) - y)^2 \mid x] &= \mathbb{E}[(h_\theta(x) - h(x) - \varepsilon)^2 \mid x] \\
&= \mathbb{E}[(h_\theta(x) - h(x))^2 \mid x] + \sigma^2 \\
&= \mathbb{E}[(h_\theta(x) - \mathbb{E}[h_\theta(x)] + \mathbb{E}[h_\theta(x)] - h(x))^2 \mid x] + \sigma^2 \\
&= \mathbb{E}[h_\theta(x) - h(x) \mid x]^2 + \text{Var}(h_\theta(x) \mid x) + \sigma^2.
\end{aligned}$$

The first term is called the ***bias*** and corresponds to a lack of capabilities of our model compared to the best predictor possible , while the second term is the ***variance*** in the estimation of θ. σ^2 is the irreducible error if y is not deterministic.

This means that when an algorithm returns a predictor sampled from a whole family, which it always does because we are sampling a dataset, there are two factors contributing to the error. The first one, the bias, tells us about the complexity of the phenomenon we are trying to predict and thus that a predictor family that is too simple will on average never make the best possible prediction. The second one, the variance, tells us that our algorithm should not output solutions that are too different from one training set to another, or we will have difficulties in finding the right predictor.

Unfortunately, more complex families tend to have higher variances (*i.e.*, they require more samples to account for the increased difficulty in choosing the right parameters) but can void the benefit they have on the bias part over simpler families.

This is called the ***Bias-Variance*** trade-off and is a crucial design question when tackling a machine learning problem. Ideally, at equal bias, we would always like to reduce the variance because it would mean less potential overfitting.

5.2 Bagging

Let us assume that we have M ***independent*** predictors h_m for the same task, and that each one of them does more than ε error with probability at most δ thanks to a PAC bound. Then, by virtue of independence, the probability that all the predictors produce an error is bounded by the product $(\delta)^M$ which is considerably better. The key factor here is the independence assumption, which is very difficult to ensure in practice. One could of course train on disjoint datasets by splitting the training set into M non-overlapping subsets, but doing so usually results in much weaker predictors than training on the entire data. Or, one could split the characteristics on which the predictors are working (*e.g.*, using non-overlapping sets of pixels in our image classification task on Bluebell), but then again the best that a predictor can achieve is severely limited.

But do we really need independence? In fact, the following proposition shows that weak correlation is all that is required to reduce the variance.

> Proposition 5.1 (Variance of correlated predictors)
> *Let $h_m(x)$ for $1 \leq m \leq M$ be M random variables of variance σ^2 and with a correlation of ρ, then the variance of the estimate $\frac{1}{M}\sum_{m=1}^{M} h_m(x)$ is $\rho\sigma^2 + \frac{1-\rho}{M}\sigma^2$ and is less than σ^2 if and only if $\rho < 1$ and $M > 1$.*

This means that a simple averaging strategy, or just vote counting in the case of classification, can decrease the variance if the predictors are not perfectly correlated. Such a strategy is often called *bagging*.

> Definition 5.2 (Bagging)
> Given a set of M predictors $h_m : \mathcal{X} \mapsto \mathbb{R}^d$ for $1 \leq m \leq M$, the bagging rule of aggregation is simply
>
> $$\bar{h}(x) = \frac{1}{M}\sum_{m=1}^{M} h_m(x).$$
>
> In the case of classification, the predicted classes can be mapped to a one hot encoding before summation, and the predicted class of the bagging aggregate is the class with the highest count.

5.2.1 *Random Forest*

In the previous chapter, we have seen that decision trees are unstable which means that the learning over this family of predictors has high variance. Bagging seems then a natural solution to get a better predictor. This idea is essentially formalized as random forest, where a randomization of the dataset is introduced to prevent the deterministic tree growing procedure to output the same model (Breiman, 2001).

Randomization can be by sampling a subset of the training set randomly, as this will change the set of possible thresholds to choose from, or it can be done by selecting a random subset of the dimensions of $\mathcal{X}$ which has the same effect. In practice, when dealing with datasets that have more dimensions than samples, it is advisable to subsample the dimensions only.

A standard algorithm for training random forests is shown below.

Algorithm 5.1 Random Forest

Input: Training set $\mathcal{A}$ of dimension d, number of trees M, number of dimensions k, maximum depth L

Output: An ensemble of M trees $\{(h_m, S_m)\}_{1 \leq m \leq M}$

for $m \leftarrow 1$ **to** M **do**
  $S_m \leftarrow$ a random subset of k dimensions among $\{1, \dots, d\}$
  $h_m \leftarrow \text{GrowTree}(\mathcal{A}|_{S_m}, 0, L)$
return $\{(h_m, S_m)\}_{1 \leq m \leq M}$

The pytorch code for the random forest using our DT class from the previous chapter is shown in Listing 5.1. Training our random forest with 400 trees of depth 10 on the Bluebell dataset reaches an average training accuracy of 100% and an average validation accuracy of 73% using cross-validation, a much stronger result than what was obtained for a single tree (38.5%).

We also show the template of the classes accumulated over the trees on Figure 5.1. The template images are much denser, which shows that the trees are using different pixels for their decision process, and we can even recognize some of the colors of the different classes.

Listing 5.1 Random Forest classifier. Trees are trained sequentially on a random subset of dimensions.

```
import numpy as np
import torch
from decisiontree import DecisionTree

class RandomForest:
    def __init__(self, n_trees=10, max_depth=3, criterion="gini", n_features="sqrt"):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.criterion = criterion
        self.n_features = n_features
    def fit(self, X, y):
        X = torch.as_tensor(X, dtype=torch.float32)
        y = torch.as_tensor(y, dtype=torch.long)
        d = X.shape[1]
```

```
        k = int(np.sqrt(d)) if self.n_features == "sqrt" else int(self.n_features)
        self.trees = []
        self.dims = []
        for _ in range(self.n_trees):
            dims = torch.randperm(d)[:k]
            tree = DecisionTree(max_depth=self.max_depth, criterion=self.criterion)
            tree.fit(X[:, dims], y)
            self.trees.append(tree)
            self.dims.append(dims)
        return self
    def predict(self, x):
        x = torch.as_tensor(x, dtype=torch.float32)
        votes = torch.stack([tree.predict(x[:, dims]) for tree, dims in
            zip(self.trees, self.dims)])
        return torch.mode(votes, dim=0).values
```

GENERALIZATION ERROR. Breiman (2001) proved that adding more and more trees to the forest does not necessarily overfit, but instead converges to the family error. Given an ensemble of predictors $h_1, \dots, h_M$, we define a margin function as

$$mg(x, y) = \frac{1}{M} \sum_{m=1}^{M} \mathbb{1}[h_m(x) = y] - \max_{j \neq y} \frac{1}{M} \sum_{m=1}^{M} \mathbb{1}[h_m(x) = j],$$

which corresponds to the difference between the true class vote and the maximum vote of a false class. The true risk is positive if and only if the margin is negative, and is given by

$$\mathcal{R}(h_1, \dots, h_M) = P[mg(x, y) < 0].$$

THEOREM 5.3 (Random forest generalization error)
As the number of trees of parameters $\theta \sim \Theta$ increases, for almost surely all sequences $\theta_1, \dots$ sampled from Θ, the generalization error $\mathcal{R}$ converges to

$$\mathbb{P}\left[\mathbb{P}_\Theta[h_\theta(x) = y] - \max_{j \neq y} \mathbb{P}_\Theta[h_\theta(x) = j] < 0\right].$$

Proof. For a fix training set and a tree θ, the prediction $h_\theta(x) = j$ corresponds to the union of hyper-rectangles, and there is only a fixed number K of such unions in the entire family parametrized by Θ. Denote those $S_1, \dots, S_K$ and define $\phi(\theta) = k$ the index function that maps a tree to a union of hyper-rectangles.

Let $N_k = \sum_{n=1}^{N} \mathbb{1}[\phi(\theta_n) = k]$ be the number of times that the S_k is used by the first N trees. Then, the average number of times that class j has been predicted by the first N trees is sum over k of the average use of the S_k,

$$\frac{1}{N} \sum_{n=1}^{N} \mathbb{1}[h_{\theta_n}(x) = j] = \frac{1}{N} \sum_{k=1}^{K} N_k \mathbb{1}[x \in S_k].$$

Since N_k/N converges to $P(\phi(\theta) = k)$ almost surely by the law of large numbers, then the union of sets for which is does not converge has zero measure. Outside of that set, we have

$$\frac{1}{N} \sum_{n=1}^{N} \mathbb{1}[h_{\theta_n}(x) = j] \to \sum_{k=1}^{K} P(\phi(\theta) = k) \mathbb{1}[x \in S_k] = P(h_\theta(x) = j).$$

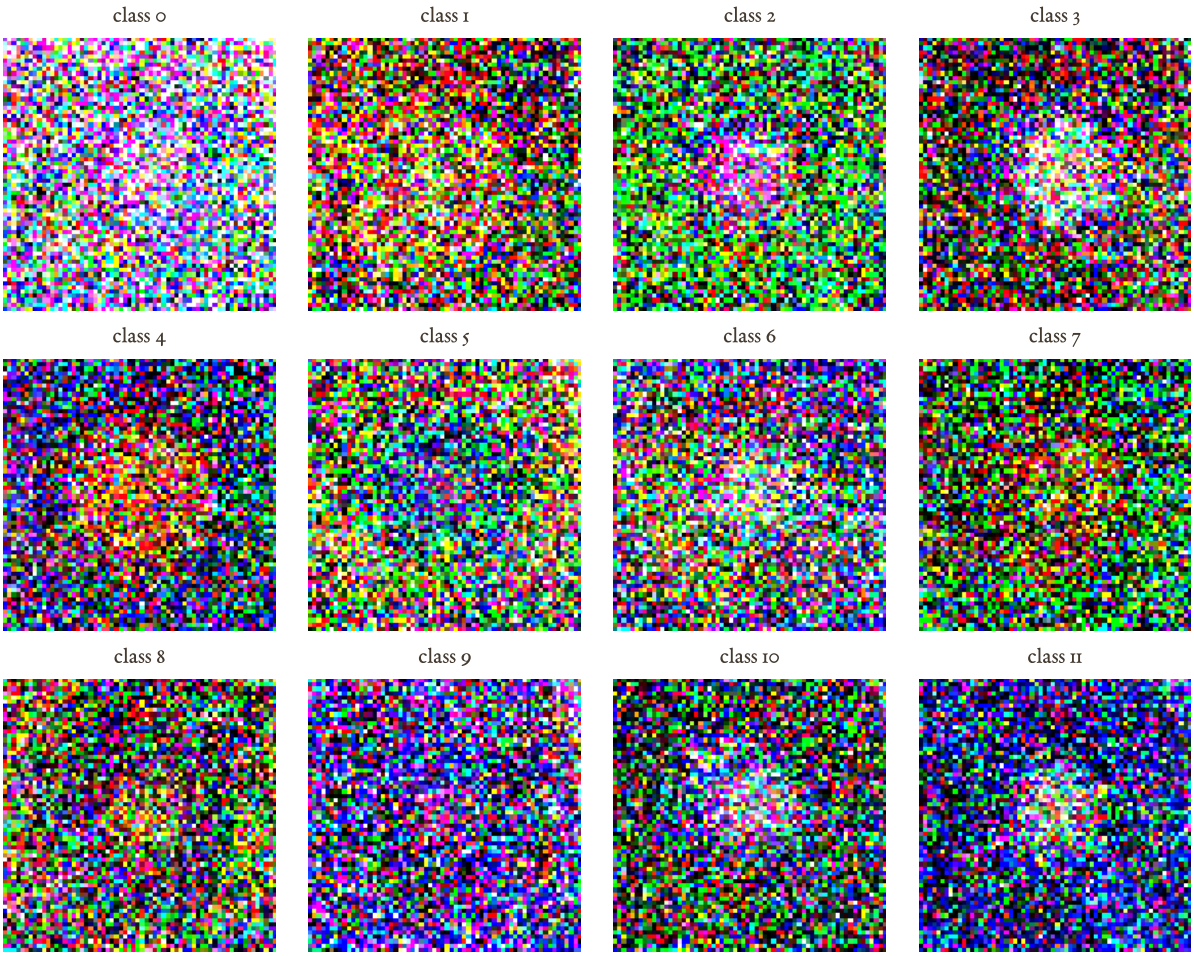

Figure 5.1 Template image derived from the RF decision thresholds on the first cross-validation split, for each of the 12 classes.

■

5.3 Boosting

In bagging, all predictors have the same weight and influence on the aggregated prediction. However, some decisions are redundant, for example in the case of easy samples that have universal agreement between predictors. Moreover, some predictors have high empirical risk and we thus know they are bad, and yet we still give them equal weight in the aggregation. Could we find a way to use this information to make a non-uniform aggregation that is either simpler or more effective? This is exactly the idea behind ***boosting***.

5.3.1 Adaboost 🐍

In (Freund and Schapire, 1996), the authors introduce the following ideas: Associate a weight to each sample and train a predictor to minimize the weighted error, then update the weights such that they increase for wrongly predicted samples or decrease otherwise and train a new classifier with the updated weight. Finally, combine the two classifiers and repeat those last operations until satisfied.

This process should produce a sequence of classifiers that increasingly focus on the ***hard*** samples as well as a way to combine them.

The key insight is to define a loss function that renders this process easy. Such function should enable both an easy computation of the weights update and easy computation of the predictors fusion. With that in mind, let us investigate the exponential loss defined below.

> DEFINITION 5.4 (Exponential loss)
> The exponential loss function applied to a predictor h for the task of binary classification is defined as
>
> $$\begin{aligned} \ell : \mathcal{X} \times \mathcal{Y} &\mapsto \mathbb{R}^+ \\ (x, y) &\to e^{-yh(x)} \end{aligned}$$

Given a classifier aggregate h, we want to add a new classifier $h_m^\star$ that reduces the error under linear combination. More formally, given a training set $\mathcal{A} = \{(x_i, y_i)\}_{1 \leq i \leq n}$, we have to solve

$$\arg \min_{\beta, h_m \in \mathcal{H}} \frac{1}{n} \sum_i \exp[-y_i(h(x_i) + \beta h_m(x_i))]$$

Notice that it can be rewritten as

$$\beta_m, h_m^\star = \arg \min_{\beta, h_m \in \mathcal{H}} \frac{1}{n} \sum_i w_i \exp[-y_i \beta h_m(x_i)],$$

with

$$w_i = \exp[-y_i h(x_i)].$$

Notice also that given $\beta > 0$

$$\arg \min_{h_m \in \mathcal{H}} \frac{1}{n} \sum_i w_i \exp[-y_i \beta h_m(x_i)]$$

is attained by

$$\arg \min_{h_m \in \mathcal{H}} \frac{1}{n} \sum_i w_i \mathbb{1}[y_i \neq h_m(x_i)]$$

because

$$\sum_i w_i \exp[-y_i \beta h_m(x_i)] = e^{-\beta} \sum_{y_i = h_m(x_i)} w_i + e^{\beta} \sum_{y_i \neq h_m(x_i)} w_i.$$

Thus, the classifier that minimizes the weighted 01-loss is also minimizing the exponential loss.

Now for β, given h_m minimizing the 01-loss, we have to solve

$$\arg \min_{\beta} \frac{1}{n} \sum_i w_i \exp[-y_i \beta h_m(x_i)].$$

We notice that

$$\sum_i w_i \exp[-y_i \beta h_m(x_i)] = \left(e^{\beta} - e^{-\beta}\right) \sum_i w_i \mathbb{1}[y_i \neq h_m(x_i)] + e^{-\beta} \sum_i w_i,$$

and thus the optimal β is given by

$$\beta = \frac{1}{2} \log \frac{1 - err_m}{err_m}, \quad err_m = \frac{\sum_i w_i \mathbb{1}[y_i \neq h_m(x_i)]}{\sum_i w_i}.$$

These results give rise to the Adaboost algorithm shown below and due to Freund and Schapire (1996).

Algorithm 5.2 AdaBoost

Input: Training set $\mathcal{A} = \{(x_i, y_i)\}_{1 \le i \le n}$, number of rounds M
Output: A classifier $h(x) = \text{sign}\left(\sum_{m=1}^{M} \beta_m h_m(x)\right)$

$\forall i,\ w_i \leftarrow 1/n$
for $m \leftarrow 1$ **to** M **do**
  $h_m \leftarrow \arg\min_{h \in \mathcal{H}} \sum_i w_i \mathbb{1}[y_i \neq h(x_i)]$
  $err_m \leftarrow \dfrac{\sum_i w_i \mathbb{1}[y_i \neq h_m(x_i)]}{\sum_i w_i}$
  $\beta_m \leftarrow \dfrac{1}{2} \log \dfrac{1 - err_m}{err_m}$
  $\forall i,\ w_i \leftarrow w_i \exp\left[\beta_m \mathbb{1}[y_i \neq h_m(x_i)]\right]$
return $h_1, \ldots, h_M, \beta_1, \ldots, \beta_M$

We show the pytorch code for this simple version of Adaboost in ??. Running it in the Bluebell dataset with trees of depth 3 and a maximum of 100 rounds, we obtain an average training accuracy of 100% and an average validation accuracy of 74.5% using cross-validation, slightly better than random forests. One caveat though, the algorithm we present is for binary classification and we thus had to wrap the predictors into a one-vs-all scheme to solve the 12-class problem of Bluebell. There exist variants with multiple classes that are much more efficient since they do not require training as many trees as we did.

Listing 5.2 Adaboost using decision trees.

```python
import torch

from decisiontree import DecisionTree

class AdaBoost:
    def __init__(self, n_rounds=10, max_depth=1, criterion="gini"):
        self.n_rounds = n_rounds
        self.max_depth = max_depth
        self.criterion = criterion
    def fit(self, X, y):
        X = torch.as_tensor(X, dtype=torch.float32)
        y = torch.as_tensor(y, dtype=torch.float32)
        n = X.shape[0]
        w = torch.ones(n) / n
        self.trees = []
        self.betas = []
        for _ in range(self.n_rounds):
            # approx weighted 01-loss by sampling using the weights
            idx = torch.multinomial(w, n, replacement=True)
            tree = DecisionTree(max_depth=self.max_depth, criterion=self.criterion)
            tree.fit(X[idx], ((y[idx] + 1) / 2).long())

```

```
            pred = tree.predict(X).float() * 2 - 1
            wrong = (pred != y).float()
            err = ((w * wrong).sum() / w.sum()).clamp(1e-10, 1 - 1e-10)
            beta = 0.5 * torch.log((1 - err) / err)

            w = w * torch.exp(beta * wrong)
            w = w / w.sum()

            self.trees.append(tree)
            self.betas.append(beta.item())
        return self
    def decision_function(self, x):
        x = torch.as_tensor(x, dtype=torch.float32)
        agg = torch.zeros(x.shape[0])
        for beta, tree in zip(self.betas, self.trees):
            agg += beta * (tree.predict(x).float() * 2 - 1)
        return agg
    def predict(self, x):
        return torch.sign(self.decision_function(x))
```

5.3.2 *Gradient Tree Boosting* 🦉

Is the exponential loss the only option to have that iterative strategy? It turns out that we can build a strategy for training a sequence of classifiers that correct the mistakes iteratively by operating a gradient descent on a function space for any loss that is twice differentiable using trees. This is the concept at the core of *Gradient Tree Boosting*. The idea is that given our sequence of classifier, we can approximate the error made at any sample x using a second order expansion and use that approximation for the next tree to decide of the splits.

More precisely, Let us consider a training set $\mathcal{A} = \{(x_i, y_i)\}$ of n samples. For a twice differentiable loss function $\ell(\cdot)$ and given the classifier $\hat{y}_i = \sum_{m=1}^{M} h_m(x_i)$, consider the regularized empirical risk

$$\hat{\mathcal{R}} = \sum_i \ell(y_i, \hat{y}_i) + \sum_{m=1}^{M} \Omega(h_m), \quad \Omega(h) = \gamma L + \frac{1}{2}\lambda \|w\|^2,$$

where L is the number of leaves in the tree and w is a vector of weights the size the number of leaves.

We can use a second-order approximation, for the round-m update $\hat{y}_i^{(m-1)}$ to $\hat{y}_i^{(m-1)} + h_m(x_i)$,

$$\hat{\mathcal{R}}_{(m)} \approx \tilde{\mathcal{R}}_m = \sum_i \left[\ell(y_i, \hat{y}_i^{(m-1)}) + G_i h_m(x_i) + \frac{1}{2} H_i h_m(x_i)^2\right] + \Omega(h_m),$$

with $G_i = \partial_{\hat{y}_i} \ell(y_i, \hat{y}_i)$ and $H_i = \partial^2_{\hat{y}_i} \ell(y_i, \hat{y}_i)$.

Remove the constant term from the approximation and define $I_l = \{i \mid q(x_i) = l\}$, the set of samples in leaf l for $1 \leq l \leq L$ using the leaf indicator function q, then

$$\tilde{\mathcal{R}}_m = \sum_{l=1}^{L} \left[w_l \sum_{i \in I_l} G_i + \frac{w_l^2}{2}\left(\lambda + \sum_{i \in I_l} H_i\right)\right] + \gamma L.$$

The optimality condition leads to an optimal weight of

$$w_l^\star = -\frac{\sum_{i\in I_l} G_i}{\sum_{i\in I_l} H_i + \lambda},$$

associated to the objective value

$$\tilde{\mathcal{R}}_m^\star = -\frac{1}{2}\sum_{l=1}^{L} \frac{\left(\sum_{i\in I_l} G_i\right)^2}{\sum_{i\in I_l} H_i + \lambda} + \gamma L.$$

This provides a criterion for splitting nodes: Letting $I = I_a \cup I_b$ the tentative split, the regularized gain is then

$$\mathcal{G} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_a} G_i\right)^2}{\sum_{i\in I_a} H_i + \lambda} + \frac{\left(\sum_{i\in I_b} G_i\right)^2}{\sum_{i\in I_b} H_i + \lambda} - \frac{\left(\sum_{i\in I} G_i\right)^2}{\sum_{i\in I} H_i + \lambda}\right] - \gamma,$$

and we split only when it is positive. Splitting can be prevented purely because of γ that acts as a regularization and prevents splitting if that makes the tree too complex. At some point, the tree ceases to grow, and we can start growing the next one, correcting the previous aggregate using a second expansion of the loss around $\hat{y}_i$.

There are very efficient implementations of many gradient boosting algorithms, notably XGBoost (Chen and Guestrin, 2016) that has been popularized as a Swiss-army knife for machine learning competition.

5.4 Exercises

Exercise 5.1 — Bias/variance of correlated trees. Derive the bias/variance decomposition of a bagged ensemble of B correlated trees, and show that variance reduction depends on the pairwise correlation between them.

Exercise 5.2 — Out-of-Bag error estimation. Implement out-of-bag (OOB) error estimation for a random forest and verify it tracks a held-out validation estimate.

Exercise 5.3 — Adaboost link to Gradient Tree Boosting. Show that AdaBoost is a special case of gradient boosting under the exponential loss

6

Neural Networks

Neural networks are the cornerstone of modern machine learning while being one of the oldest learning algorithm proposed. In this chapter, we investigate the basics of neural networks up to the architectures that launched the ***deep learning*** revolution. We try to characterize fundamental properties of neural networks as well as the main concepts that are necessary to understand why deep learning was so successful.

6.1 Formal neuron and Perceptron

The formal neuron model has been proposed by McCulloch and Pitts (1943) using symbolic logic. In this model, the state of a neuron is binary (activated or not) and the excitation of the neuron depends only if a sufficient number of synapses are excited while no inhibitory synapse is active. This can be modeled by a linear combination with a weight vector w, a bias b and a thresholding function σ, leading to the following equation

$$h(x) = \sigma(\langle w, x \rangle + b),$$

that is not explicitly mentioned in the original paper.

We show the usual graphical representation in Figure 6.1. In this figure, we imply the input x, the weight vector w and the bias b are part of any vector space, but the original paper is concerned with boolean algebra and formalize neural networks as a form of logic circuit. Interestingly, they use results from logic and what would become known later ***symbolic AI*** to investigate what types of proposition, and thus ultimately functions, can be encoded by networks of such neurons, depending on the existence of cycles.

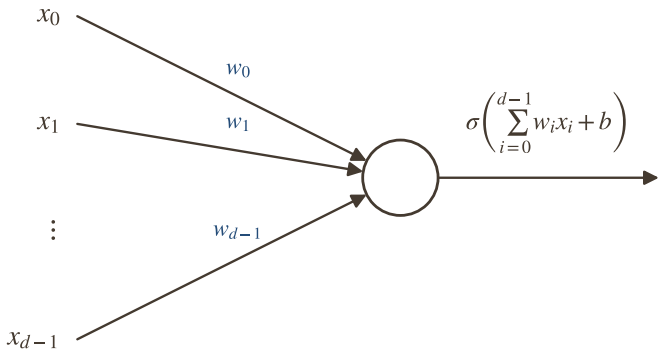

Figure 6.1 The formal neuron: a weighted sum of the inputs $x_0, \ldots, x_{d-1}$ plus a bias b, passed through an activation function σ.

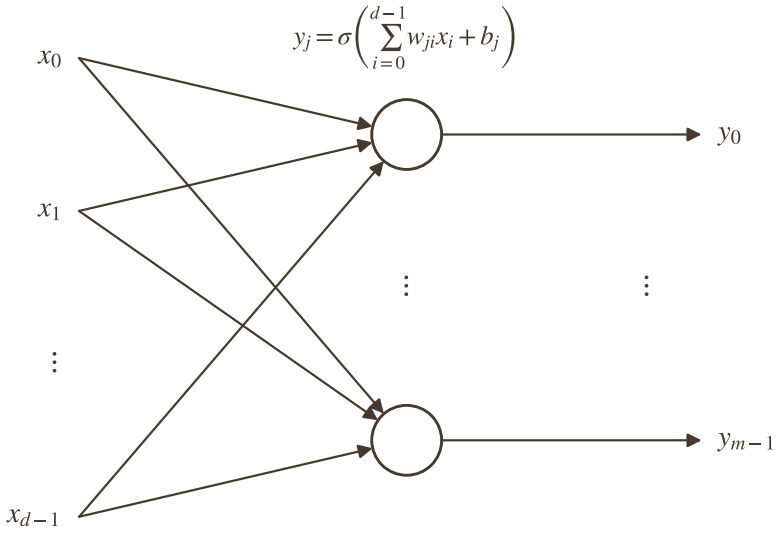

Figure 6.2 A single-layer perceptron: d inputs are fully connected to m neurons, each producing its own output y_j.

6.1.1 Perceptron

If we assume that $x \in \mathbb{R}^d$ instead of being a boolean vector, then a single neuron looks like a linear model with an activation function σ, but its origins are more rooted in circuits than statistics. In fact, the ***Perceptron***, arguably the first neural network, by Rosenblatt (1958), was first and foremost built as a circuit. In that circuit, sensory inputs are mapped to outputs using the sum and threshold strategy, and several output units are considered, similarly to what is shown on Figure 6.2.

In the original paper, the outputs are linked to one another via inhibition signals such that only one can be active at a time. Rosenblatt does not propose a formal learning algorithm in the sense of machine learning, but instead provides rules for updating the circuit. These rules are ***Hebbian*** in inspiration, after the Canadian psychologist Donald O. Hebb who proposed a theory of biological learning, and increase the weights that are connected to both an active input and output. This reinforces the capability in that input to activate the corresponding output, while being totally

unsupervised. This leads to random association of outputs to specific input patterns. To perform actual supervised learning, the Perceptron is modified such that by default nothing happens, but an operator can manually activate an output and launch the update so as to force the reinforcement of the corresponding weights.

In modern times however, the Perceptron simply refers to a linear model with differentiable function and that is trained using gradient descent on a typical loss like the ℓ_2 error (similar to linear regression) or a cross-entropy (then equivalent to logistic regression).

6.1.2 Impossibility theorem

The main limitation to the perceptron is the same as for linear prediction, notably linear classification, which is that most problems are not linearly separable. The following theorem puts these limitations clearly.

> THEOREM 6.1 (Perceptron impossibility)
> ***Given a sample space** $\mathcal{X} = \mathbb{R}^d$ **and a label space** $\mathcal{Y} = \{0, 1\}$**, there exists a distribution on pairs** $(x \in \mathcal{X}, y \in \mathcal{Y})$ **that cannot be learned by a predictor of the form** $h(x) = \sigma(w^\top x + b)$ **with** σ **a strictly increasing activation function.***

Proof. Let us consider the XOR problem on $\mathbb{R}^2$ where vector $(1, 0)$ and $(0, 1)$ map to 1 and vectors $(0, 0)$ and $(1, 1)$ map to 0. Let θ be the value such that $\sigma(\theta) > k$ is considered as an output of 1 and $\sigma(\theta) \leq k$ is considered an output of 0. Because σ is strictly increasing, there exists only one of such value.

Assume the XOR problem can be solved by our linear predictor with the activation function σ. Then, satisfying all four cases of the XOR problem is equivalent to satisfying the following equations,

$$\begin{aligned} (0,0): &\quad b \leq \theta \\ (0,1): &\quad w_2 + b > \theta \\ (1,0): &\quad w_1 + b > \theta \\ (1,1): &\quad w_1 + w_2 + b \leq \theta \end{aligned}$$

Summing equations 1 and 4 and comparing it to the sum of equations 2 and 3 leads to $w_1 + w_2 + 2b$ being both greater and lesser or equal than θ, which is a contradiction. ■

6.2 Multiple Layer Perceptron

To overcome the limitations of the perceptron-like architecture, we have to abandon quasi-linear methods. This is essentially the same step that we made with kernels, by projecting our data using a non-linear mapping with the hope that the problem is linearly separable in that space. The neural network answer is to build that non linear mapping from stacking Perceptrons on top of one another benefiting from the non linearity between them, as shown in Figure 6.3. The resulting neural network is called a ***Multiple Layer Perceptron*** or MLP for short.

More formally, a layer f_i of an MLP is the function that maps to $\mathbb{R}^{d_i}$ by stacking linear predictors with activation functions (or neurons):

$$f_i(x) = [\sigma(w_{ij}^\top x + b_{ij})]_{j \leq d_i},$$

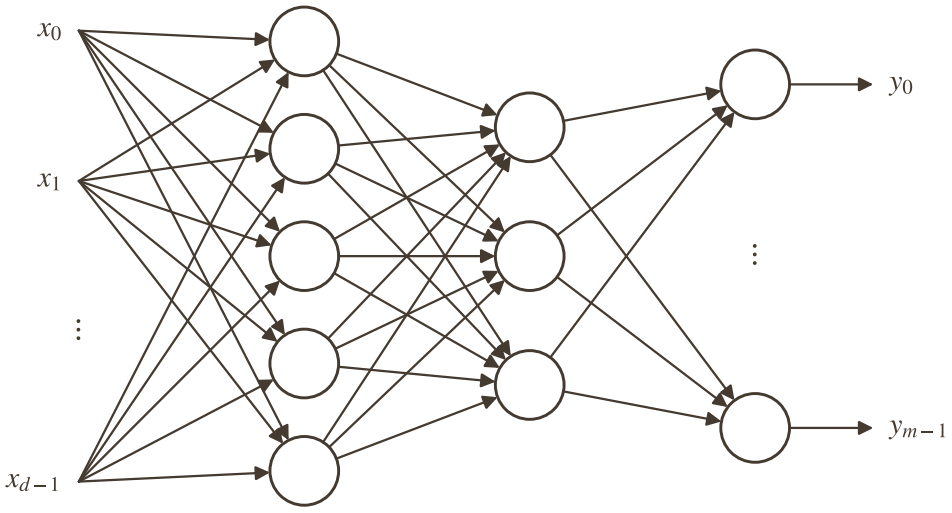

Figure 6.3 A multilayer perceptron: the input x is passed through 3 fully-connected layers before producing the output y.

and the MLP is the composition of L layers

$$F(x) = f_L \circ \cdots \circ f_1(x).$$

6.2.1 Activation functions

We list below some of the activation functions that are commonly used in neural networks as well as their properties. The corresponding plots are shown in Figure 6.4.

Sigmoid The sigmoid activation function, defined as

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

has a range between 0 and 1 and is mostly used as a gating function by multiplying it to other outputs, the equivalent of a binary AND. It saturates at both ends, which means that its gradient is mostly flat which can prevent good training.

Hyperbolic tangent The hyperbolic tangent, defined as

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

has output between -1 and 1. It behaves similarly to the sigmoid, but with an extended range.

ReLU Rectified Linear Units, introduced by Nair and Hinton (2010), are one of the simplest non-linear activation function, and are defined as

$$\sigma(x) = \max(0, x).$$

ReLU were introduced to overcome the gradient problems of sigmoid and hyperbolic tangent activations, but having a gradient equal to the identity on its right part. The main issue is that the left part leads to a ***dead unit*** as the gradient is null. Thus a neurons that is always in the left part is ***dead*** in the sense that it does not contribute to the network, and is unrecoverable through gradient descent.

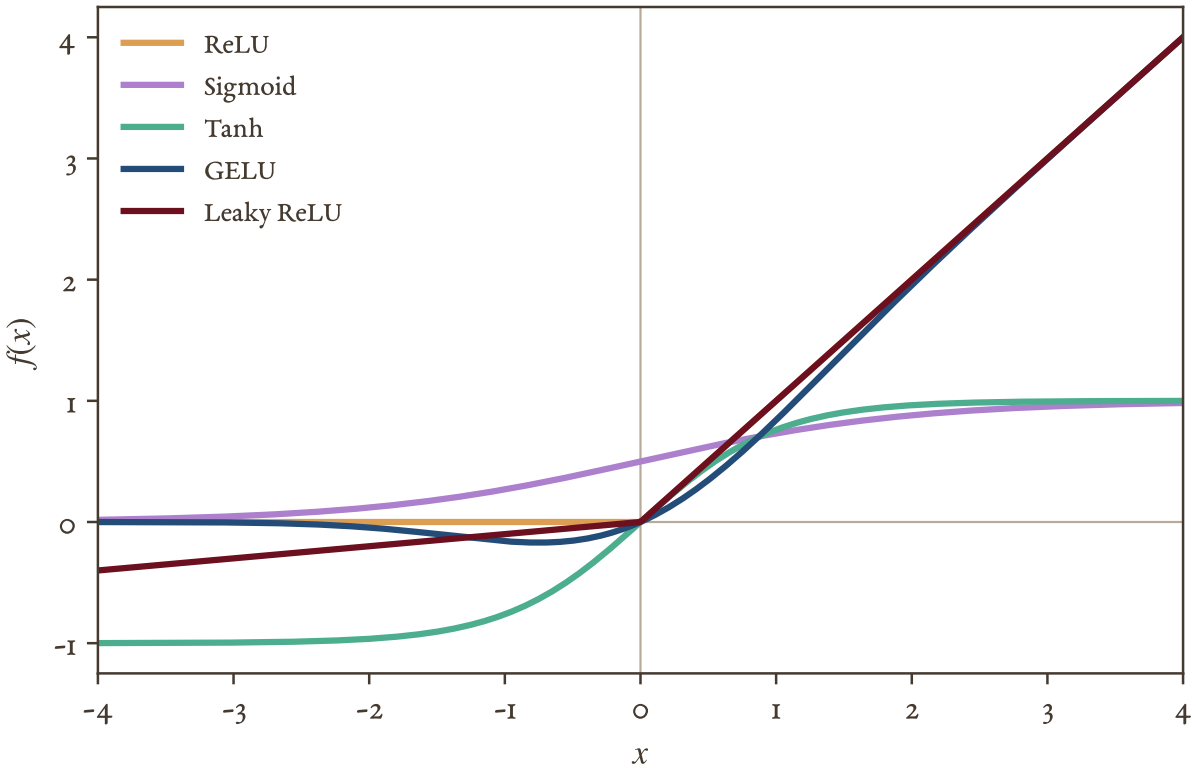

Figure 6.4 Common activation functions used in neural networks.

Leaky ReLU Leaky ReLU introduced by Maas et al. (2013) are meant to solve the problem of dead units in ReLU. They are defined by

$$\sigma(x) = x \text{ if } x > 0 \text{ else } -\alpha x,$$

with typically $\alpha = 0.01$. The left part now has a gradient and the function is still globally non-linear.

GeLU Gaussian Error Linear Unit, proposed by Hendrycks and Gimpel (2016) is defined as

$$\sigma(x) = x\Phi(x),$$

with Φ the cumulative distribution function of the normal distribution. It is smooth and differentiable everywhere, and has no dead neurons since its gradient is 0 only at a value that contributes to the network.

6.2.2 Gradient Back-propagation

To train an MLP, the common strategy is to use gradient descent following the ERM principle. Given a training set $\mathcal{A} = \{(x_i, y_i)\}$ of size n, we want to minimize the empirical risk

$$\hat{\mathcal{R}}_{\mathcal{A}}(F) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, F(x_i)),$$

over the set of parameters w_l for all layers $1 \leq l \leq L$ in F.

The gradient of layer l is estimated using a mini-batch strategy. For example for the weights of layer l, we compute

$$\frac{1}{B}\sum_i \frac{\partial \ell(y_i, F(x_i))}{\partial w_l},$$

with B the size of the mini-batch.

To estimate the gradient at any particular layer, an automatic differentiation engine is used. There are many algorithms to compute the gradient in a composite function, and the most efficient are implemented in popular deep learning libraries like `pytorch` or `JaX`. Early neural networks were optimized using back-propagation which consists of the following recursion

$$\begin{aligned}
\delta_L &= \ell'(y, F(x)) \circ \sigma'(x_L) \\
\delta_k &= w_{k+1}^\top \big(\sigma'(x_{k+1}) \circ \delta_{k+1}\big) \\
\frac{\partial \ell(y, F(x))}{\partial w_k} &= \delta_k \, x_{k-1}^\top \\
\frac{\partial \ell(y, F(x))}{\partial b_k} &= \delta_k
\end{aligned}$$

where x_k is the output of layer k and is stored during the forward pass. These relations can be obtained from the optimality condition of the Lagrangian that considers the constrained problem where the output of a layer $f_k(w_k, x_k, b_k)$ has to match the input x_{k+1} of the next layer, as shown by LeCun (1988).

The pseudo-code for a very naive back-propagation algorithm is given below. It has been shown that back-propagation is at most 3 times as costly as the forward pass. We must admit that we are lucky to live in a universe where one can minimize the empirical risk using differentiation that is cheap to compute with a universal recipe instead of a universe that would require us to compute costly integrals!

Algorithm 6.1 Back-propagation

Input: Input x, target y, weights $w_1, \dots, w_L$ and biases $b_1, \dots, b_L$
Output: Gradients $\frac{\partial \ell(y, F(x))}{\partial w_k}$ for $1 \leq k \leq L$

$x_0 \leftarrow x$
Forward pass
for $k \leftarrow 1$ **to** L **do**
    Compute and store $x_k \leftarrow f_k(w_k, x_{k-1}, b_k)$
Backward pass
Compute $\ell'(y, F(x))$
$\delta_L \leftarrow \ell'(y, F(x)) \circ \sigma'(x_L)$
for $k \leftarrow L - 1$ **to** 1 **do**
    $\delta_k \leftarrow w_{k+1}^\top \big(\sigma'(x_{k+1}) \circ \delta_{k+1}\big)$
for $k \leftarrow 1$ **to** L **do**
    Update w_k and b_k using $\ell'(y, F(x)), \delta_k, x_{k-1}$
return $\left\{\frac{\partial \ell(y, F(x))}{\partial w_k}\right\}_{1 \leq k \leq L}$

By looking carefully, we can immediately see a problem with the back-propagation algorithm: if the gradients at all layers are $|\partial| < 1$, then their product vanishes to 0, which is called the ***vanishing gradient*** problem ; if they are $|\partial| > 1$, their product grows in absolute value creating instabilities in the update called the ***exploding gradient*** problem. This prevents us from training very deep architectures.

6.2.3 Universal Approximation theorems

The main appeal of MLPs is that not only do they solve the XOR problem (see **??**), but they can also solve any problem. This property is called ***universal approximator*** and is backed by numerous theorems.

> THEOREM 6.2 (Universal Approximation for the sign function)
> *$\forall n$, the family of MLP of depth 2 (with a single hidden layer) equipped with the sign activation function contains all functions from $\{\pm 1\}^n$ to $\{\pm 1\}$.*

Proof. Consider a network with a single hidden layer of 2^n neurons. Let $\{u_i\}_{1 \leq i \leq n}$ be the set of k input vectors that have label 1. Notice that $\forall i, \langle u_i, u_i \rangle = n$ and $\forall x, \forall i, x \neq u_i \Leftrightarrow \langle x, u_i \rangle \leq n - 2$ as there is at least a 2-bit difference between different binary vectors. Set k neurons to $h_i(x) = \text{sign}(u_i^\top x - n + 1)$, we have $h_i(x) = 1$ iff $x = u_i$ and -1 else. Set the output to $F(x) = \text{sign}(\sum_i h_i(x) + k - 1)$. ■

There were many of such theorems developed in the end of the 1980s , most notably by Hornik et al. (1989) and Cybenko (1989), qualifying the type of function that can be approximated with what type of activation. This should not come as a surprise. There are many decompositions that can approximate continuous functions to arbitrary precision if you allocate enough budget on the components, like wavelets for example. Our proof is essentially memorizing the entire set of samples with label 1, which would require a gigantic number of neurons as the training set grows. Unfortunately this is the case for shallow networks. A single hidden layer MLP requires an exponential number of neurons to model what a deep network would do linearly.

> THEOREM 6.3 (Universal Approximation requires depth)
> *Let h be a neural network with ReLU activation, $2k$ layers and $O(k)$ neurons that oscillates 2^k times across its midpoint. For any single-hidden-layer ReLU network of width w to preserve the same number of topological oscillations across the function's midpoint, its width must satisfy*
>
> $$w \geq 2^k.$$

Proof. This is an application of Theorem 1.1 by Telgarsky (2016) that holds similar results for other activations than ReLU and even linear combinations of Decision Trees. ■

Using an MLP with a single hidden layer, we can solve the same 2D problem that we used for the k-NN and the decision trees. We train an MLP with 8 neurons on the hidden layer and show the output in Figure 6.5. The output is a combination of the hidden layers and produces a decision boundary that is mostly correct.

6.3 CONVOLUTIONAL NEURAL NETWORKS

Many problems in machine learning have input samples for which the key patterns that decides the label are not always on the same dimensions of the input. This is the case for temporal signals, because

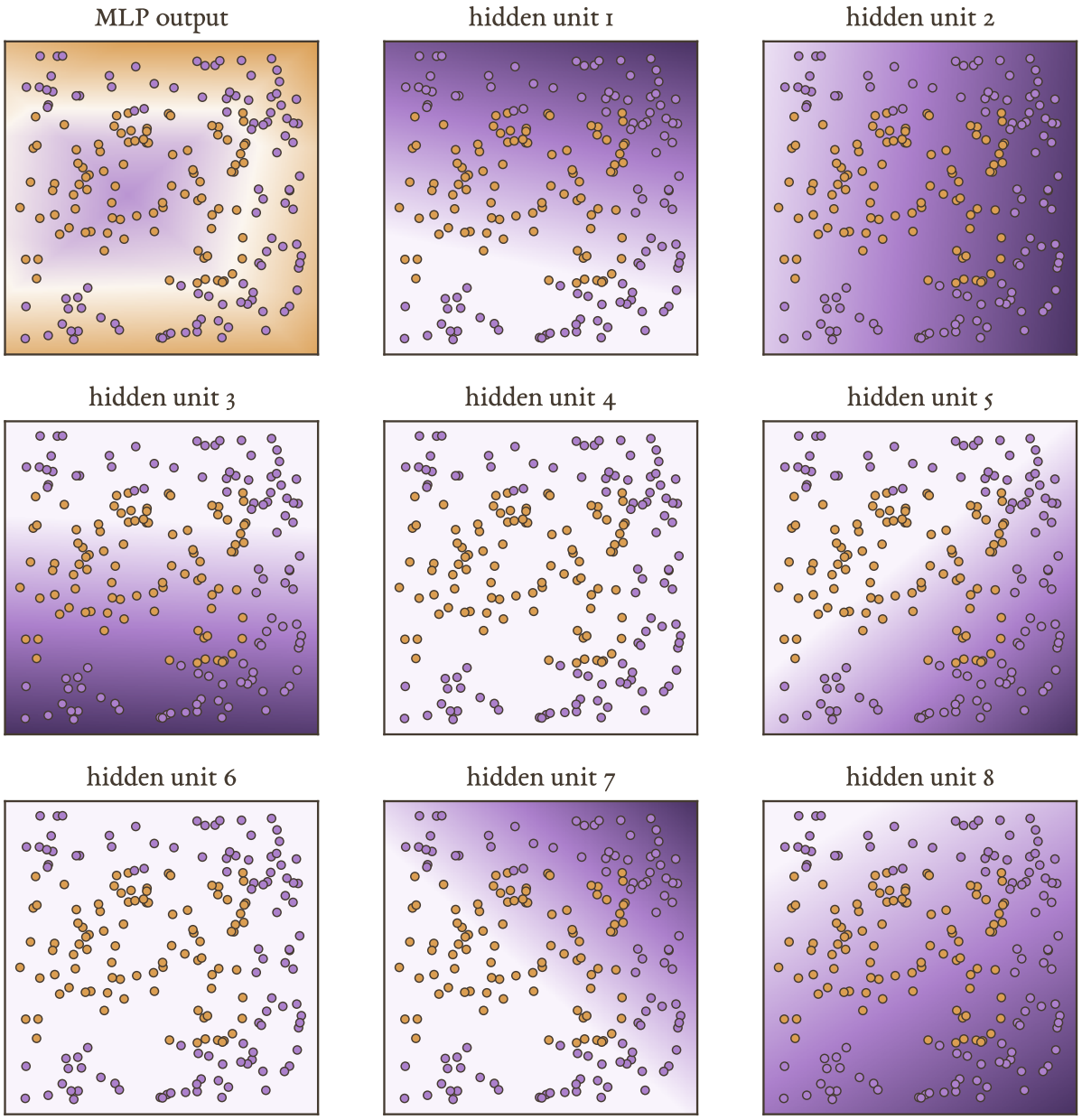

Figure 6.5 An MLP with a single hidden layer of 8 ReLU units solving the rectangle problem. Top-left: the MLP's final (linear) output, showing a piecewise-linear decision boundary assembled from the hidden units' own boundaries. Remaining panels: each hidden unit's own (post-ReLU) response, each contributing one linear crease.

the event of interest can happen at various time. This happens also in image recognition because the pattern of interest does not necessarily have a fixed position in the image. In our Bluebell dataset, the flowers are never perfectly aligned, although some visual pattern are characteristic.

The proper way to handle this is to have layers that are ***equivariant*** to specific transforms.

> Definition 6.4 (Equivariance)
> A function h is equivariant with respect to an operator T if
>
> $$\forall x, h(Tx) = Th(x).$$

In the case of temporal signals or images, translation equivariance is what we are after, which is achieved by convolution.

> Definition 6.5 (Convolution)
> In neural networks, the convolution between a discrete sequence $x[n]$ and the convolution kernel h is
>
> $$(h \star x)[n] = \sum_{k=-\infty}^{\infty} x[n+k]h[k].$$

The convolution in neural networks is a slight abuse of language in that it should be really called ***cross-correlation*** owing to the lack of reflection of the kernel, but it is not an important distinction as there is no practical difference between the two.

> Proposition 6.6 (Convolution is translation equivariant)
> *Let $x[n]$ be a discrete sequence, h a convolution kernel and T a translation operator, then*
>
> $$(h \star x)[Tn] = T(h \star x)[n]$$

The translation equivariance proposition is trivial to prove from the definition, but it holds only for infinite signals. In the case of finite signals, such as image, boundary effects kick in and allow the network to distinguish pattern from their absolute position.

For signals with several spatial axes, the convolution operation is extended over these too. For example, for $2D$ images $x[m, n]$ and a kernel h, we get

$$(x \star h)[m, n] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} x[m+k, n+l]h[k, l],$$

which is a $2D$ cross-correlation rather than a $2D$ convolution.

Color images have 3 channels, red, green and blue, and thus $x[m, n]$ is a 3-dimensional vector. We could perform a convolution over that axis too, but it does not make much sense as the channel dimensions are arranged arbitrarily and thus the $m + k$ that encodes a ***neighboring*** aspect is void. Instead, we consider a vector kernel $h[m, n] \in \mathbb{R}^3$ as well and use the inner product in place of the multiplication,

$$(x \star h)[m, n] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} \langle x[m+k, n+l], h[k, l] \rangle,$$

which can be used for arbitrary dimensions beyond 3. To retain the dimension, we can perform several of such convolutions and stack them together. More formally, if we want a $3D$ convolution that goes from d dimensions to p dimensions, we need to define a convolutional kernel $h[m, n] \in \mathbb{R}^{d \times p}$, and

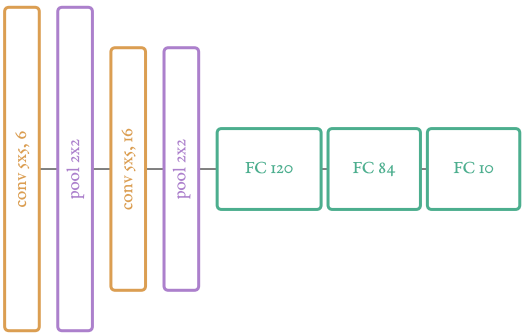

Figure 6.6 LeNet-5.

the convolution operation becomes

$$(x \star h)[m, n] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} h[k, l]^\top x[m + k, n + l],$$

which is the core working principle of $2D$ convolutional layers used in convolutional neural networks.

6.3.1 LeNet

The first convolutional neural network (CNN) was proposed by LeCun et al. (1998) for the recognition of handwritten digits in zip codes. The architecture is shown in Figure 6.6. It is a small network, related to the computing power that was available in the early 90s. Nonetheless, it sets the principles of a CNN, which are that convolutional are interleaved with downsampling operation so as to increase the spatial support of the kernel as the layers progress, while the number of neurons per layer is increased so as to retain more and more discriminative information. At the end, the image is composed of a single pixel with many channels, which is equivalent to a vector, and a structure analog to an MLP can take over until the decision is output.

6.3.2 AlexNet & VGG

The main CNN breakthrough was achieved by Krizhevsky et al. (2012) with the AlexNet architecture shown in Figure 6.7. The architecture is very similar to that of LeNet, but extended to fit the maximum computation cost available on a GPU at the time, leading to 60M parameters. This CNN won the ImageNet 2012 competition (a classification problem with 1000 classes and 1.2M training images) and it was a shock for all the computer vision community who has been working on handcrafted image features to feed to kernel machines for years. The improvement was so significant that it halted research on handcrafted features after a year or two.

AlexNet was almost impossible to reproduce due to its GPU engineering. The first CNN reaching state-of-the-art performances was VGG developed in Oxford by Simonyan and Zisserman (2015). The architecture is extended to contain more convolutional layers so as to reduce the cost of the fully connected layers at the end. They also provided a Matlab toolkit, Matconvnet, that allowed researchers to quickly test ideas around CNN. VGG was also demonstrating the wall of training deep CNN with older techniques, as it is about the maximum depth that could be trained in one go. Beyond that depth, vanishing gradients prevent an easy training and we have to resort to sequentially increasing

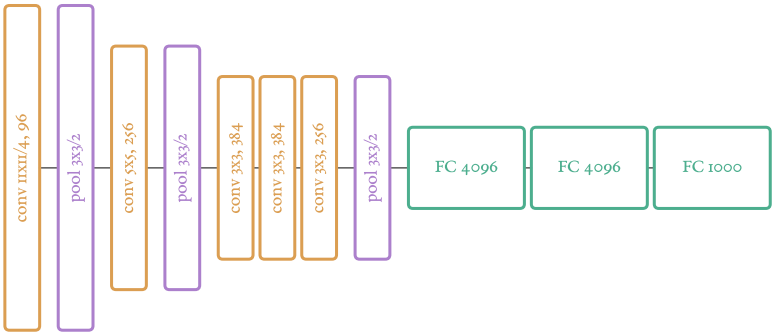

Figure 6.7 AlexNet.

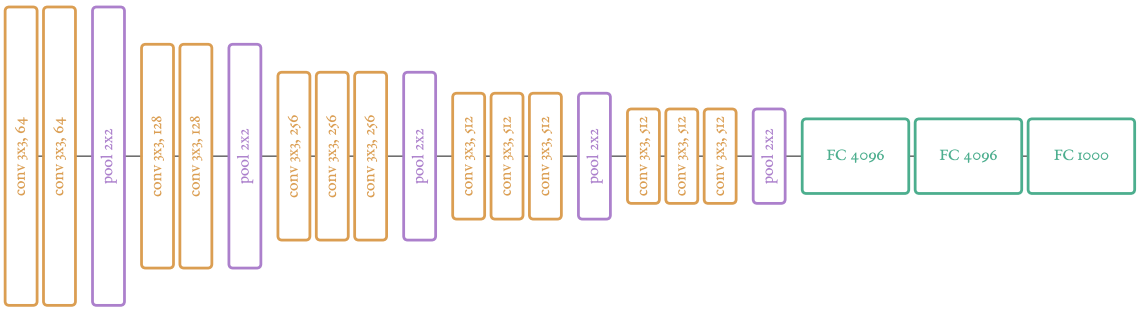

Figure 6.8 VGG-16.

the depth after supervising shorter models and ***baby-sitting*** the training (***i.e.***, saving intermediate checkpoints to restart training with a lower learning rate if instabilities appear, ***etc***).

6.3.3 ResNet

The most popular CNN architecture is without any doubt ResNet by He et al. (2016). It solves the vanishing gradient problem by introducing residual connections such that every layer only learns a difference from the previous one,

$$x_{l+1} = h_l(x_l) + x_l,$$

which means that the gradient is always the identity plus the layer's derivative, which prevents vanishing. This residual framing is now ubiquitous in modern neural network architectures, and is probably the most important contribution in recent deep learning. With the limited space available on paper, we only show in Figure 6.9 the ResNet-34 architecture that has 34 layers, although the ResNet-50 with 50 layers is probably the more popular one. In the original paper, the authors even train an architecture with 1000 layers to demonstrate the effectiveness of the residual connections against vanishing gradients.

A small ResNet with 18 layers trained on Bluebell leads to an average train accuracy of 98.1% and an average validation accuracy of 93.3% on the cross-validation five splits. Such architecture has 14M parameters. This should alarm us: it is clearly over-parametrized with more than 1000 parameters per training image, and yet it achieves the best results so far with the least overfitting we have seen. Why

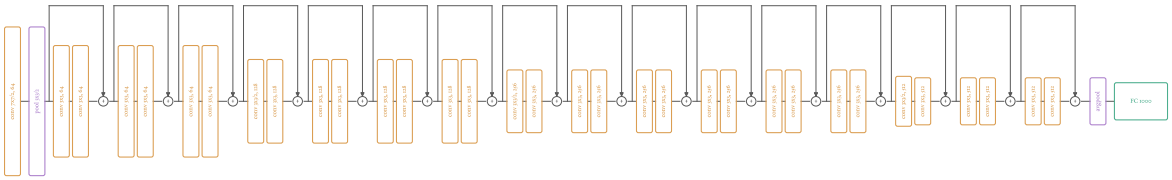

Figure 6.9 ResNet-34. Each "+" marks where a residual shortcut is added back into the main path.

does it not overfit? Is the SRM principle wrong? The next chapter, chapter 7, will try to answer these questions.

As an example, the `pytorch` code for our small ResNet is shown in **??**.

, label=lst:resnet]shared/code/smallresnet.py

6.4 Exercises

Exercise 6.1 — XOR weights by hand. Using a single hidden layer MLP with only 2 neurons on the hidden layer and equipped with the sign activation, compute by hand the weights that solve the XOR problem.

Exercise 6.2 — MLP impossibility other than XOR. Prove the perceptron impossibility theorem for a function other than XOR.

Exercise 6.3 — Activation dead unit comparison. Compare ReLU, Leaky ReLU, and GELU on a MLP and measure the fraction of dead units over training for each.

Exercise 6.4 — Convolution equivariance for finite signals. Using the equivariance definition, construct a concrete boundary example showing that convolution and translation don't commute for finite (as opposed to infinite) signals.

7

Deep Learning

Deep Learning comes at the realization that machine learning is so effective, it should be used for the entire process instead of relying on human knowledge and intuition to design (***handcraft***) features to feed to the learning machine. This is essentially what CNN proposed. Instead of designing visual features that we think are discriminative for vision problems, we let the learning algorithm find them.

This realization has led to the first AI revolution of the 21st century, starting from the ImageNet moment with AlexNet winning to the release of the first ***Large Language Model*** which started the second AI revolution of the 21st century centered around Generative AI. Contrarily to the belief that massive data and increasingly access to computational resources are solely responsible for the progress, these two revolutions have led to many modernizations of the training of neural networks that would not have made the results possible in the 80s even assuming that the data and the computational resources were available, which they were obviously not.

In this chapter, we aim at providing some of the tools that have made deep learning so effective and reliable, as well as some tentative explanations as to why it works despite the SMR principle being very much not confident about it.

7.1 Modern training

Modern training of deep learning architectures relies on massive models with millions if not billions (trillions even!) of parameters. Training these models requires obviously access to large computational resources, but it is not enough. We show here three aspects of training a deep network that are now standard.

7.1.1 Regularization

We have seen in chapter 2 that regularization prevents overfitting. Neural networks are not totally immune to the effect, in particular in the case of smaller training sets that could lead them to memorize spurious associations between intermediate features.

Weight decay. When using a loss that does not vanish, like the categorical cross-entropy, there is always some gradient even for correctly predicted samples. As such, the weights that are correlated with the input features will be reinforced as long as the training occurs. This leads to inflated weight that can either disrupt the training or worse, prevent learning other correlations which have activations of much lower magnitude.

To prevent that, the common strategy is to perform weight decay,

$$w_i \leftarrow (1 - \lambda_w) w_i,$$

for all weight w_i and a very small value λ_w. This corresponds to an $\ell_2(w_i)$ regularization of the weights, analog to the ridge that we have used in chapter 2.

One has to be careful with the interaction of weight decay and optimizer, in particular if weight decay is implemented as an additional loss instead of a direct optimization.

Dropout. Another way to prevent spurious associations between a neuron and an input feature is to simply randomly remove input features to force the neuron to rely on other signals. This is the idea behind dropout proposed by Srivastava et al. (2014). During training, the layer f_i takes a modified input

$$x_{i+1} = f_i(x_i \circ a_i),$$

with a_i a binary variable of the same size as x_i sampled from the Bernoulli distribution with probability $1 - p$ (*i.e.*, a probability p of being dropped). At test time, keeping the dropout creates variability in the output. Turning it off requires adapting the input to prevent a gap between the training and test input distributions,

$$x_{i+1} = f_i((1 - p)x_i),$$

such that the input is the same in expectation. Alternatively, that rescaling can be done during training to avoid the extra cost during test.

One can also leave the dropout active during test, which means that the output is now a random variable and varies if we query the network several times. This can simulate a pseudo-ensemble that allows to measure the variability in the network's output depending on the sample being processed. Samples that have a high variability indicates that the network is relying on very specific but rare features to compute the decision, and may be sensitive to noise.

The pytorch code for Dropout (with rescaling during training) is shown in Listing 7.1.

7.1.2 Normalization

Due to positive feedback effect in the back-propagation algorithm (a weight that has a strong positive effect on the decision has its magnitude increased), intermediate features can grow in magnitude which makes the training possibly unstable. Normalizing the input of a layer is a way to prevent outputs with high magnitude and thus reduce the effect of such feedback loop.

Listing 7.1 Dropout as a PyTorch module, inverted-dropout convention: elements are zeroed with probability p and the survivors rescaled by $1/(1-p)$ at train time, so no rescaling is needed at test time.

```python
import torch
import torch.nn as nn

class Dropout(nn.Module):
    def __init__(self, p=0.5):
        super().__init__()
        self.p = p  # drop probability
    def forward(self, x):
        if self.training:
            mask = torch.bernoulli(torch.full_like(x, 1 - self.p))
            return x * mask / (1 - self.p)
        return x
```

Batch Normalization. Batch normalization, BatchNorm for short, was introduced by Ioffe and Szegedy (2015) to reduce what the authors call the *covariate shift*. Training layer f_i using back-propagation makes it adapt itself to recognize meaningful features in the distribution of its input x_i, but it also changes x_i through the optimization of f_{i-1} *after* it has changed f_i, creating a gap between the distribution of the input at the time of optimizing f_i and after optimizing f_{i-1}.

To reduce this effect, the authors propose to normalize the input such that its mean and variance are fixed. They introduce an operation performed on the input before the it goes into the layer,

$$\mathrm{BN}(x_i) = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

where μ and σ^2 are the mean and variance of x_i, β and γ are trainable parameters intended to preserve a learned offset and scale for the signal, and ϵ is for numerical stability.

To estimate the statistics μ and σ^2, they propose an exponential moving average (EMA) strategy using the batch statistics,

$$\mu \leftarrow \eta\mu + \frac{1-\eta}{B}\sum_n x_i[n],$$
$$\sigma^2 \leftarrow \eta\sigma^2 + \frac{1-\eta}{B}\left(\sum_n x_i^2[n]\right) - \frac{1-\eta}{B^2}\left(\sum_n x_i[n]\right)^2,$$

with $\eta = 0.999$ typically.

These EMA statistics are evolving slowly compared to the training process (stability being the whole point of the normalization), and thus cannot be used during training. In place, the batch statistics are used. This creates a discrepancy between training and test since both are using different statistics. In particular, it prevents training with a small batch size because the batch statistics would be too noisy.

In addition, the EMA is often very much adapted to the training set and creates a discrepancy with test samples. In the case where test samples have a small domain gap (*e.g.*, colored images *vs* black-and-white images), updating the ema on the test data (which does not require supervision) can greatly improve the performances.

Listing 7.2 Batch normalization as a PyTorch module: batch statistics at train time, an exponential moving average (η) tracked for use at eval time.

```python
import torch
import torch.nn as nn

class BatchNorm(nn.Module):
    def __init__(self, n_features, eps=1e-5, eta=0.999):
        super().__init__()
        self.eps, self.eta = eps, eta
        self.gamma = nn.Parameter(torch.ones(n_features))
        self.beta = nn.Parameter(torch.zeros(n_features))
        self.register_buffer("running_mean", torch.zeros(n_features))
        self.register_buffer("running_var", torch.ones(n_features))
    def forward(self, x):
        # x: (batch, n_features)
        if self.training:
            mean, var = x.mean(dim=0), x.var(dim=0, unbiased=False)
            with torch.no_grad():
                self.running_mean = self.eta * self.running_mean + (1 - self.eta) *
                    mean
                self.running_var = self.eta * self.running_var + (1 - self.eta) * var
        else:
            mean, var = self.running_mean, self.running_var
        x_hat = (x - mean) / torch.sqrt(var + self.eps)
        return self.gamma * x_hat + self.beta
```

The pytorch code for a BatchNorm module is shown in Listing 7.2

Layer Normalization. The dependency on the batch size of batch normalization being very limiting in many applications, other normalizations schemes were developed. We present here LayerNorm, but other types are popular, and have different names depending on the axis on the input tensor that is normalized.

More formally, LayerNorm processes a sample x_i at layer i such that

$$\mathrm{LN}(x_i) = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

with μ and σ^2 the estimated mean and variance of the sample computed over its last dimension. For example, if x_i is a sequence of vectors, then the statistics are computed independently for each vector and applied as such. If x_i is an image, the normalization is performed independently for each pixel. Notice that γ and β are tensors of the same size as μ and σ^2 and that the multiplication is element-wise.

The learnable affine transform is optional, but it often helps the network train better. Indeed, both β and γ could be absorbed in the weights of the surrounding layers. Separating this rescaling and offset operation makes the other layer focus on their own feature transform without having to handle the scale of the features themselves, whereas mixing the two could results in conflicting gradients that optimize neither of those.

Listing 7.3 Layer normalization as a PyTorch module: statistics computed per sample over the last dimension, so no running estimate is needed.

```python
import torch
import torch.nn as nn

class LayerNorm(nn.Module):
    def __init__(self, n_features, eps=1e-5):
        super().__init__()
        self.eps = eps
        self.gamma = nn.Parameter(torch.ones(n_features))
        self.beta = nn.Parameter(torch.zeros(n_features))
    def forward(self, x):
        # x: (..., n_features); statistics computed over the last dimension only
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, unbiased=False, keepdim=True)
        x_hat = (x - mean) / torch.sqrt(var + self.eps)
        return self.gamma * x_hat + self.beta
```

The pytorch code for a LayerNorm module is shown in Listing 7.3

ADAPTIVE LAYERNORM. More recently LayerNorm has been used as a way to incorporate auxiliary information into the flow of transformations performed by the network. For example, in the case of image generation (which we will see with more details in **??**), giving information about the type of object that one wants to generate make the generation controllable.

In that case, the learnable parameters γ and β are the output of another neural network (typically a small MLP) that takes this auxiliary signal as input. This allows the full model to bias the main signal and solves a different problem than the covariate shift of BatchNorm.

7.1.3 Optimizers

Another big improvement achieved by deep learning over the vanilla training of neural networks is with respect to optimizers.

SGD WITH MOMENTUM. When training deep neural networks, the number of trainable parameters is often sufficiently larger than the number of samples that could fit in a mini-batch of a stochastic gradient descent that is limited by the computational resources available. Some of these parameters are responsible for detecting rare features in the data. The estimation of their gradient is thus very noisy, and they get rarely updated.

An easy way to increase the effective batch size is to accumulate gradients over several batches before performing the update pass. Indeed, since the gradient is just summed over the samples of the batch, nothing prevents doing the sum sequentially. This strategy is called *gradient accumulation* in the literature and is often used in very large models. However, it is inherently slow because the updates are less frequent than with a smaller but fully parallel batch.

Instead, a common strategy is to buffer the past gradients using an exponential moving average, a method often called SGD with *momentum* because it intuitively corresponds to some inertia in

the weight. Following the newtonian dynamics analogy, the gradient becomes a force that updates a velocity v_θ of the weight w_θ, and the weights are updated by their velocity,

$$v_\theta \leftarrow m v_\theta + g,$$
$$w_\theta \leftarrow w_\theta - \eta v_\theta,$$

with g the gradient estimated with a mini-batch strategy and η the learning rate.

The algorithm in pseudo-code is given below.

Algorithm 7.1 SGD with momentum

Input: Loss L, training set $\mathcal{A} = \{(x_i, y_i)\}$, initial parameters $\mathbf{w}_0$, learning rate η, momentum m, batch size B, number of steps T

Output: Trained parameters $\mathbf{w}$

$\mathbf{w} \leftarrow \mathbf{w}_0$
$v \leftarrow 0$
for $t \leftarrow 1$ **to** T **do**
  Sample a minibatch $\mathcal{A}_t \subset \mathcal{A}$ of size B
  $g \leftarrow \nabla_{\mathbf{w}} \frac{1}{B} \sum_{(x_i, y_i) \in \mathcal{A}_t} L(\mathbf{w}, x_i, y_i)$
  $v \leftarrow mv + g$
  $\mathbf{w} \leftarrow \mathbf{w} - \eta v$
return $\mathbf{w}$

Notice that the accumulation of the gradient into the velocity changes the magnitude of the update. With a decay rate of m and assuming that all gradient have the same magnitude and sum in the same direction, the velocity's magnitude goes to $1/(1-m)$ as the number of steps grows. As such, the learning rate has to be updated consequently to avoid overshooting the stability region of SGD.

To give a better understanding of how an optimizer in implemented in deep learning libraries, we show in Listing 7.4 a pytorch implementation of SGD with momentum. In that code, the gradient has been computed during the `backward` call and is stored with the parameters, whereas the velocity is stored within the optimizer. All operations are done ***in place*** (***i.e.***, changing the variable at its current location in memory) so as to not allocate new intermediate variables. By convention in many libraries, in place operations are denoted with a "_" at the end, like "`mul_`" line 17.

ADAM. While SGD with momentum was able to train large neural networks that participated in the deep learning revolution, it is very slow for larger networks. The reason is that big residual networks tend to have big regions of rather flat landscape where the gradient is small, and by accumulation into the momentum buffer, this makes the optimization process slow down.

Ideally, we would like the opposite behavior, slowing down where the loss is steep so as to not overshoot and slow down when the loss is flat to compensate for the lack of change. This is typically what second order methods do by taking into account the curvature of the loss landscape, but computing the Hessian of a million-parameter neural network is computationally infeasible. Instead Adam, proposed by Kingma and Ba (2015), tracks the square of the gradient and normalizes the step size by its inverse such that when the magnitude of the gradient decreases we take larger steps to compensate for it. This introduces a second exponential moving average with a parameter β_2 that has

Listing 7.4 SGD with momentum as a PyTorch optimizer: the velocity accumulates the raw gradient, geometrically discounted by m, and is what actually updates the parameters.

```python
import torch

class SGDMomentum(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-2, momentum=0.9):
        super().__init__(params, dict(lr=lr, momentum=momentum))
    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            lr, m = group["lr"], group["momentum"]
            for p in group["params"]:
                if p.grad is None:
                    continue
                state = self.state[p]
                if "velocity" not in state:
                    state["velocity"] = torch.zeros_like(p)
                v = state["velocity"]
                v.mul_(m).add_(p.grad)
                p.add_(v, alpha=-lr)
```

to be tuned in relation to the exponential moving average β_1 of the gradient as shown in the algorithm below.

Algorithm 7.2 Adam

Input: Loss L, training set $\mathcal{A} = \{(x_i, y_i)\}$, initial parameters $\mathbf{w}_0$, learning rate η, decay rates β_1, β_2, numerical stability ϵ, batch size B, number of steps T

Output: Trained parameters $\mathbf{w}$

$\mathbf{w} \leftarrow \mathbf{w}_0$
$m \leftarrow 0, v \leftarrow 0$
for $t \leftarrow 1$ **to** T **do**
  Sample a minibatch $\mathcal{A}_t \subset \mathcal{A}$ of size B
  $g \leftarrow \nabla_{\mathbf{w}} \frac{1}{B} \sum_{(x_i, y_i) \in \mathcal{A}_t} L(\mathbf{w}, x_i, y_i)$
  $m \leftarrow \beta_1 m + (1 - \beta_1) g$
  $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$
  $\hat{m} \leftarrow m / (1 - \beta_1^t)$
  $\hat{v} \leftarrow v / (1 - \beta_2^t)$
  $\mathbf{w} \leftarrow \mathbf{w} - \eta \hat{m} / (\sqrt{\hat{v}} + \epsilon)$
return $\mathbf{w}$

Adam is the *de facto* standard optimizer for training neural networks. Ironically, the proof of convergence in the original paper was flawed and Reddi et al. (2018) showed convex counter-examples

Listing 7.5 Adam as a PyTorch optimizer: per-parameter running estimates of the gradient's first and second moment, bias-corrected to account for their zero initialization. Weight decay omitted for simplicity.

```python
import torch

class Adam(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8):
        super().__init__(params, dict(lr=lr, betas=betas, eps=eps))
    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            lr, (beta1, beta2), eps = group["lr"], group["betas"], group["eps"]
            for p in group["params"]:
                if p.grad is None:
                    continue
                state = self.state[p]
                if "t" not in state:
                    state["t"] = 0
                    state["m"] = torch.zeros_like(p)
                    state["v"] = torch.zeros_like(p)
                state["t"] += 1
                t, m, v = state["t"], state["m"], state["v"]
                m.mul_(beta1).add_(p.grad, alpha=1 - beta1)
                v.mul_(beta2).addcmul_(p.grad, p.grad, value=1 - beta2)
                m_hat = m / (1 - beta1 ** t)
                v_hat = v / (1 - beta2 ** t)
                p.addcdiv_(m_hat, v_hat.sqrt().add_(eps), value=-lr)
```

for which Adam does not converge. Under additional conditions, typically $0 \leq \beta_1 < \beta_2 < 1$, Adam converges (locally for non-convex problems such as optimizing deep neural networks) as shown by Défossez et al. (2022).

We show in Listing 7.5 the pytorch code for Adam. Notice that the improvement in convergence speed at a memory cost since there are now 3 times the number of parameters to store for the optimizer: the gradient, the gradient momentum and the gradient square momentum. This can pose a problem for very large networks where the optimizer takes a substantial amount of GPU memory which can then fit only smaller batch sizes. Care should also be taken when resuming optimization as Adam requires some steps to get its buffers well behaved at initialization. In the case of resuming the optimization, it is best to save the buffers (the optimizer state) such that the buffers are already well behaved.

Learning rate schedule. The last piece of modern optimization of deep neural networks is the learning rate schedule. The dilemma when using a constant learning rate is that it is fast at the beginning but very much unstable towards the end. This can be solved by a decaying learning rate which stabilizes training towards the end. Several learning rate schedulers have been investigated over the years, such as stair-case decay (in the original ResNet paper), linear decay, square-root decay, but

Listing 7.6 Cosine learning rate schedule with linear warmup: the learning rate ramps up linearly over `warmup_steps`, then decays following a cosine curve down to `min_lr` by `total_steps`.

```python
import math
import torch

class CosineWithWarmup(torch.optim.lr_scheduler.LRScheduler):
    def __init__(self, optimizer, warmup_steps, total_steps, min_lr=0.0):
        self.warmup_steps = warmup_steps
        self.total_steps = total_steps
        self.min_lr = min_lr
        super().__init__(optimizer)

    def get_lr(self):
        t = self.last_epoch
        if t < self.warmup_steps:
            scale = t / max(1, self.warmup_steps)
        else:
            progress = (t - self.warmup_steps) / max(1, self.total_steps -
                self.warmup_steps)
            scale = 0.5 * (1 + math.cos(math.pi * min(progress, 1.0)))
        return [self.min_lr + (base_lr - self.min_lr) * scale for base_lr in
            self.base_lrs]
```

ultimately, a cosine decay seems to be the most successful for training very large neural networks. We show a pytorch code for such a scheduler in Listing 7.6. It includes a warmup phase where the learning rate is increased linearly as this has also been shown empirically to be stable for higher peak learning rates leading to faster optimization.

The main issue with the cosine scheduler is that it requires knowing in advance the total number of training steps. This is often a problem when developing a new model, but as we will see at the end of this chapter, there are *scaling laws* that predicts the optimal trade-off in training compute budget which can thus be used to set the number of steps.

7.2 Over-parametrization

The success of deep learning may come as a surprise to the reader who followed carefully the elements of learning theory we introduced in the beginning of this textbook. We mentioned that generalization guarantees can only be given for rather simple learning machines (*e.g.*, low VC dimension) and yet the recent results seem to be that bigger neural networks work better. How do we solve the apparent paradox?

We first want to recall that generalization bounds are *guarantees of success* only, and not proofs of failure, that is, if we adhere to the conditions, then we get a guarantee on the error rate, but if we do not, we just have no idea what will happen. We could be lucky and still have a good predictor. Or it even could be that no luck is required and that going bigger has a good probability of success. This is

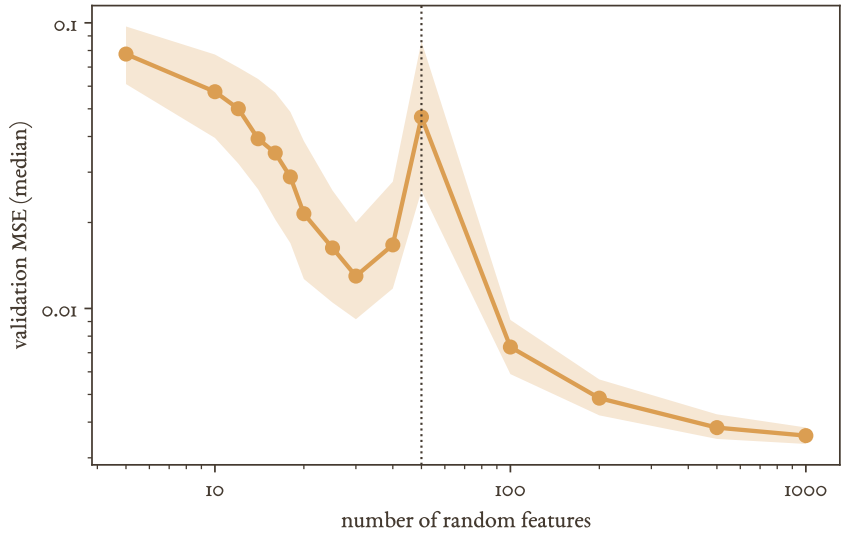

Figure 7.1 Double descent for random Fourier features ridge regression on a well-specified teacher: the target is itself a random linear combination of D_{teacher} random Fourier features, fit with D random features by closed-form ridge regression. Validation error (median over repeats, shaded band: 16th–84th percentile) first decreases as D grows and the fit's bias shrinks, spikes sharply at the interpolation threshold $D = n_{\text{train}}$ (dotted line) where the min-norm solution's variance blows up, then decreases again in the overparametrized regime.

what we investigate here.

7.2.1 Double Descent

Belkin et al. (2019) showed a phenomenon of ***double descent*** when increasing the complexity of a predictor family. We replicate a synthetic example of this phenomenon in Figure 7.1, that consists in generating samples using a large combinations of random Fourier features (see subsection 3.4.2) in a low dimensional space. We generate in dimension 15 with 100 random features for the label, and we have only 50 training samples, but suppose we do not know that the groundtruth has this high number of non-linear features. We train a non-linear ridge regression on this dataset, with an increasing number of random Fourier features. In that example, the learning problem looks very hard: it is somewhat low-dimensional, but very non-linear and we do not have a lot of training samples. Using 30 random features, twice the number of input dimensions, looks like the best SRM deal and indeed the validation loss increases after that. One could be tempted to stop there as the validation loss increases after that point and it also looks silly to add many more effective dimensions than the input.

If we do it nonetheless, the loss attains a peak and then decreases again, showing the ***double descent*** phenomenon. The peak is exactly when we have as many dimensions as the number of training samples and is called the ***interpolation threshold***. This happens because the minimum norm solution in that case consists in fitting the training set perfectly, and we cannot trade lower norm without doing additional error. When we add more Fourier features, we have more degrees of freedom and the solution is no longer unique. We can achieve a lower norm without doing additional error by spreading the energy on more components. This is what makes the solution more robust and thus generalizes better.

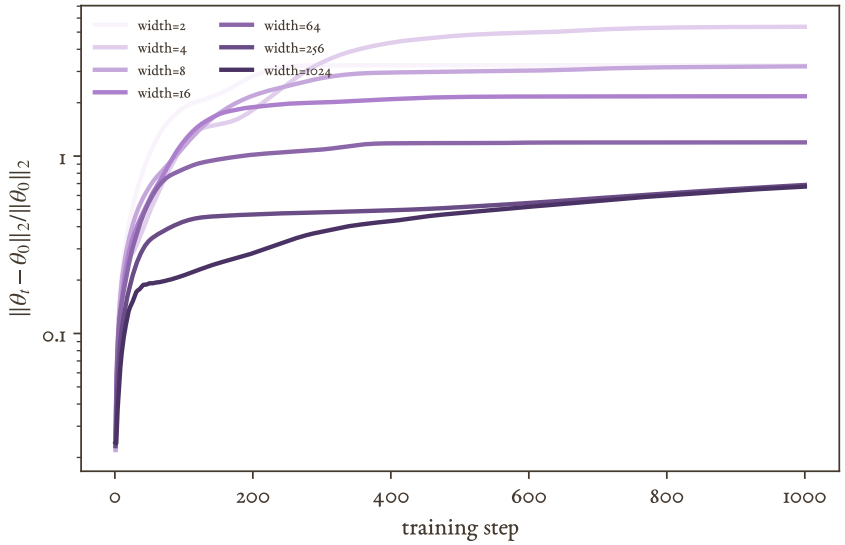

Figure 7.2 Relative distance traveled by the parameters during training, $\|\theta_t - \theta_0\|_2 / \|\theta_0\|_2$, as a function of training step. In the lazy (NTK) regime, wide networks barely move from their initialization even as the loss converges.

7.2.2 Lazy training - NTK

To investigate the effect of optimization on larger models, we use the example of chapter 6 and vary the number of neurons on the hidden layer. We know that we need at least 3 neurons to solve the problem, corresponding to the three intersecting lines that delineate one class from the other, but how does the training go if we have many more neurons than required? We show the distanced traveled by the weights during training in Figure 7.2, normalized by the norm of the initialization. Quite surprisingly, larger networks travel less while being much more successful. This seems to indicate that larger networks are easier to optimize from a weight-modification point of view.

Jacot et al. (2018) proposed a kernel interpretation of this phenomenon. Consider the loss function as a function of the weights w instead of of the sample x,

$$l_x(w) = l(y, F_w(x)),$$

and approximate it using its Taylor expansion

$$l_x(w) \approx l_x(w_0) + \nabla l_x(w_0)^\top (w - w_0).$$

This corresponds to a linear model with the non-linear mapping

$$\phi(x) = \nabla l_x(w_0),$$

defining a kernel, called the ***Neural Tangent Kernel***. This approximation tends to be better as the width grows.

Notice that some weights are highly influential ($|\sum_i \phi(x_i)[j]| \gg 0$), while others are blind spots in the kernel ($\sum_i \phi(x_i)[j] \approx 0$) and their corresponding components are barely updated.

This interpretation says that very large model are easier to train because only few neurons need to be changed to fit the training data.

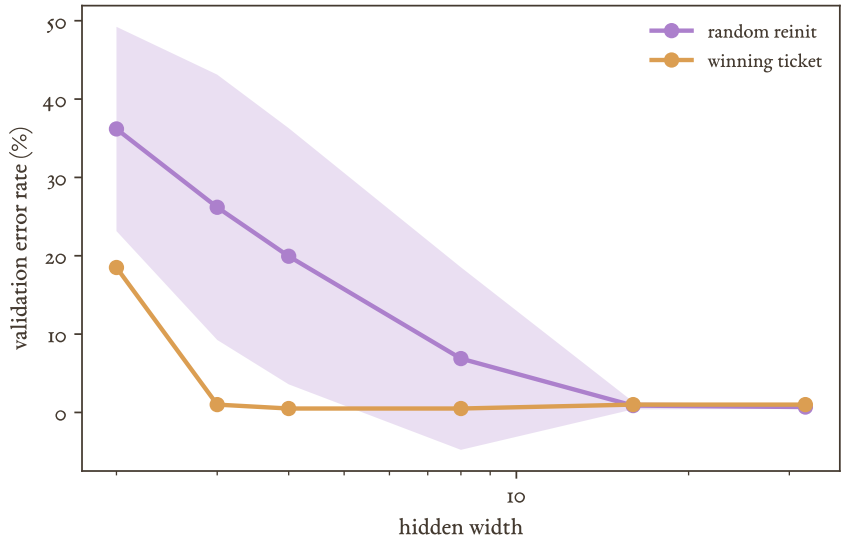

Figure 7.3 Lottery Ticket Hypothesis on the rectangle problem: a width-64 MLP is magnitude-pruned down to each width on the x-axis, with surviving hidden units reset to their original initial values (winning ticket) rather than retrained from scratch. Compared against a same-width network given a fresh random reinitialization (mean ± 1 std over 8 seeds). Below the ~3-hyperplane capacity this problem needs, the winning ticket keeps working while random reinitialization degrades sharply and becomes highly variable.

7.2.3 Lottery ticket

Could it be that training a very large neural network with most of the weights barely moving is just trusting our luck that we get at least a few parameters correctly initialized? This is essentially the *Lottery Ticket Hypothesis* proposed by Frankle and Carbin (2019). It states that a randomly-initialized dense neural network contains a sub-network that is initialized such that, when trained in isolation, it can match the test accuracy of the original network after training for at most the same number of iterations.

Inspired by Malach et al. (2020), we test this hypothesis on the same dataset as we tested the width and investigate whether we can find such a small sub-network. We know the problem can be solved with only 3 neurons on the hidden layer. Thus, we train a network with 64 neurons on its hidden layer and keep the final model as a reference. Such model obtains excellent results. We use the magnitude of the weights after convergence as a survival criterion for pruning the initialized network, that is, we keep only the neurons with highest converged magnitude initialized at their original initialization, and we train this smaller network. Results are shown in Figure 7.3. As we can see, this strategy keeps the validation performances unchanged until keeping only 3 of the original neurons, the bare minimum we know is required to solve the problem.

To show the difference between this *winning ticket* and a random width-3 network, we train a hundred of these and show the validation error density plot in Figure 7.4. Randomly initializing a small network presents two modes: one successful one where the initialization was close enough to let the optimizer discover the 3 boundaries necessary to solve the problem, and a second mode that fails, presumably because the initialization did not provide a neuron close enough to one of the 3 required boundaries and instead, 2 neurons were made redundant by focusing on the same boundary.

This provides a combined interpretation for the success of very deep architectures. By having over-parametrized networks, the initialization is provided with more coins to put in the slot-machine

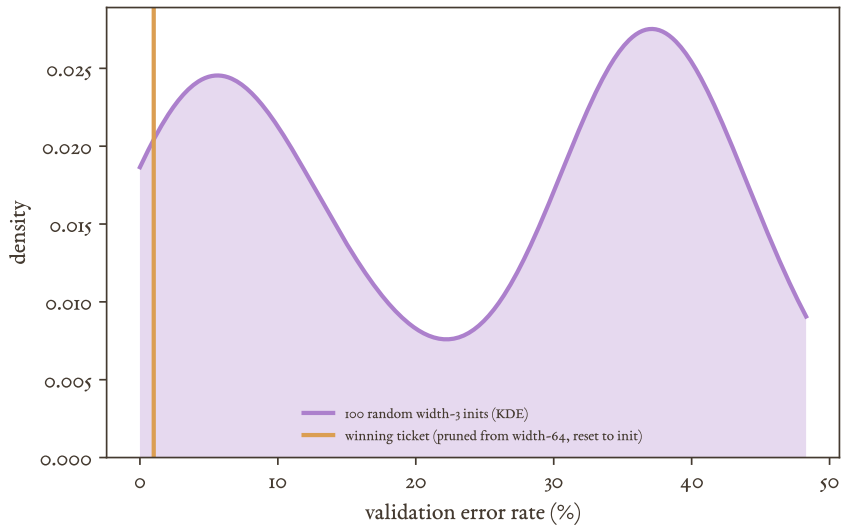

Figure 7.4 Distribution (kernel density estimate over 100 independent random initializations) of validation error for a width-3 MLP – just enough capacity to solve the rectangle problem, so most random initializations fail to find a good solution. The winning ticket's initialization (dashed line) reliably lands in the low-error mode instead of needing a lucky random draw.

returning useful neurons for the task. By NTK, neurons that are not useful are not optimized and they do not perturb the optimization process. Neurons that are redundant participate in the double descent phenomenon that prevents overfitting.

7.2.4 Scaling Laws

The question of how large a neural network should be is the final question that we have to ask. We can always increase the size of the network, but it will require more computing resources to train and at some point, it may cross the interpolation threshold depending on how much data we have. Is the increased cost worth it in terms of performances? It would be nice to have a predictive tool that could help us design the best trade-off between training cost and performances.

This idea has been pioneered by Kaplan et al. (2020) for large language models and tries to fit a power law on the validation loss against design choice hyper-parameters such as the size of the model, the size of the dataset, the training compute budget, or even the learning rate and the batch size. These empirical models have been extremely successful in predicting where to spend an increased budget to get the most performances.

We show an example on our Bluebell dataset in Figure 7.5. In chapter 6, we trained a small ResNet that achieved the best performances so far on this dataset. Here, we train variant that have reduced width, which decreases the total training cost as measured by total number of floating point operations performed (FLOP). We can see that the best performance achieved by a model from this family is well fitted by a power law of the total FLOP, and that smaller models are more efficient at smaller budgets than larger models. For a larger budget, continuing optimizing a smaller model is a waste of resources and one should instead increase the model size.

These graphs are obviously very costly to generate since they require running many networks over several random seeds to get a clean picture. However, empirical scaling laws are usually fairly robust for a few orders of magnitude past the last point which means that it is often better advised to spend a

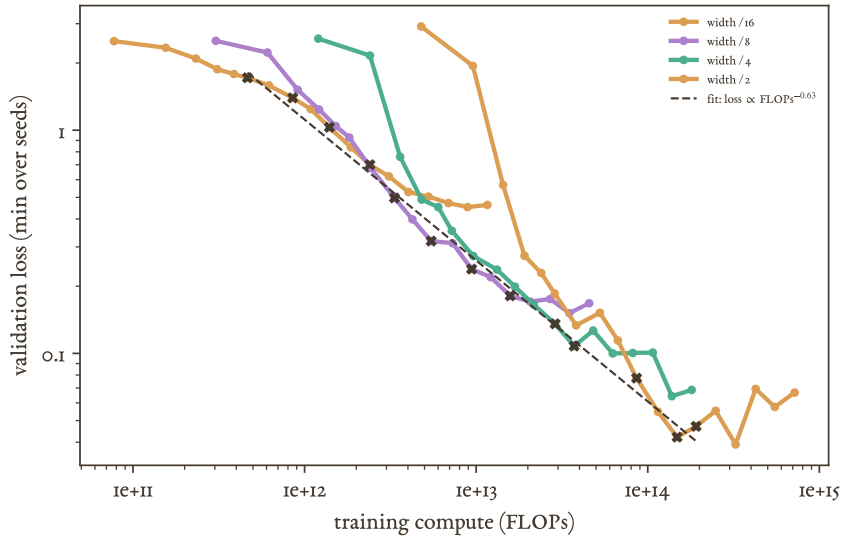

Figure 7.5 Scaling law for SmallResNet on Bluebell: validation loss against cumulative training compute (FLOPs) for the network at $1/16$, $1/8$, $1/4$, $1/2$ and full width, each curve the minimum validation loss over 30 seeds at each of a log-spaced ladder of training checkpoints. The dashed line is a power-law fit to the compute-optimal frontier (marked with ×): the best loss reached by *any* width at a given compute budget.

significant portion of the budget making them and then having a single large model training that is successful, than trying 2 or 3 shots at random for the large model directly.

7.3 Exercises

Exercise 7.1 — Weight decay and Adam. Explain why weight decay is not exactly equivalent to ℓ_2 regularization once you're using Adam rather than plain SGD.

Exercise 7.2 — Mini scaling law. Fit your own mini scaling law on a toy problem. Train 3 widths of a small MLP across a range of compute budgets, plot the compute-optimal frontier, and check whether a larger model matches the extrapolated trend.

8

Sequence modeling

So far, we have tackled static tasks with predictions that are made independently and in no particular order. The world is, however, a dynamical system that changes with the arrow of time and making predictions in arbitrary order is not a realistic case. For example, most signals have a temporal aspect to them, like text or audio.

In this chapter, we investigate how deep learning as transformed sequenced modelling by training deep neural networks that process information in a sequential manner.

8.1 Vanilla Recurrent Neural Networks

We want to model sequence-to-sequence mappings with a neural network. Suppose there is sequences of pairs $\{x_t, y_t\}_{0 \leq t \leq T}$ with some causal ordering, that is, y_t depends only on $x_{\leq t}$, and we want to train a predictor that outputs y_t. In the simplest case, $y_t = x_{t+1}$ which corresponds to a unique sequence that unfolds ***auto-regressively***.

We could use a neural network to model the transition from x_t to y_t,

$$y_t = f_\theta(x_t),$$

but such mapping is Markovian (memory-less) and does not model long delayed interaction. For example, the auto-regressive sequence $1, 1, -1, -1, 1, 1, \ldots$ is impossible to approximate with such model because the same input ($x_t = 1$) can lead to different outputs ($y_t = 1$ or -1).

Instead, a ***recurrent neural network*** summarizes the input sequence into a hidden state h_t that is kept for the next prediction, hence the name recurrent, modeled as

$$\hat{y}_{t+1}, h_{t+1} = f_\theta(x_t, h_t),$$

with $\hat{y}_{t+1}$ the prediction, and the hidden state h_t initialized to $h_0 = 0$.

We can train such model by unrolling the forward pass, ***i.e.***, by following the following algorithm:

Algorithm 8.1 RNN unrolling

Input: Input sequence $x_0, \dots, x_T$, RNN f_θ
Output: Output sequence $y_0, \dots, y_T$, hidden state sequence $h_0, \dots, h_t$

$h_0 \leftarrow 0$
for $t \leftarrow 1$ **to** T **do**
  $\hat{y}_{t+1}, h_{t+1} \leftarrow f_\theta(x_t, h_t)$
return $\{\hat{y}_t\}, \{h_t\}$

This means that unfortunately, the forward pass of an RNN on an input sequence has to be performed sequentially, which can be very costly for very long sequences.

In the case of an auto-regressive process, ***i.e.***, $y_{t+1} = x_t$, we simply train the network with a shifted input sequence as its target,

$$\min_\theta \sum_t \ell(x_{t+1}, f_\theta(x_t, h_t)_{\hat{y}),}$$

which is called ***teacher forcing*** because the model always sees the groundtruth input instead of its own prediction. This creates a discrepancy between the training and the test distributions since any small error is likely to propagate and make it such that the network will operate inputs (x_t, h_t) that have never been seen during training.

If we consider a linear model,

$$h_{t+1} = W_h^\top h_t + W_x x_t,$$
$$\hat{y}_{t+1} = W_o h_{t+1},$$

for some hidden matrix W_h and input matrix W_x, then the unrolling is

$$h_{t+1} = W_h^t W_x x_0 + g(x_1, ..., x_t),$$

for some function g that accumulates the other hidden states. This unrolling makes it apparent that depending on the spectrum of W_h, h_{t+1} could either vanish or explode. Usually, RNNs have a bounded non-linearity (such as a `tanh`) and the explosive side is contained. However, the vanishing is a real problem that is not easy to solve as it primarily comes from the limited capacity of a fixed-size hidden state. There is only so much information that we can put in h_t and after some time, we have to forget some information about the full sequence.

We show in Listing 8.1 a very simple RNN in pytorch, with the forward unrolling. The backward is handled automatically by pytorch, which each call to the sub-modules automatically adding to their respective gradients.

8.1.1 Universal approximation theorems

There are several universal approximation theorems for RNN, which can take many different aspects. For example, Siegelmann and Sontag (1992) showed that RNN can simulate any Turing machine with a linear time cost penalty of the original machine. A more interesting result is the following universal approximation theorem by ichi Funahashi and Nakamura (1993) that states that any trajectory, solution

Listing 8.1 A simple RNN as a PyTorch module, with a separate readout head: the hidden state recurrence $h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1})$ is kept distinct from the prediction $y_t = W_{\text{ro}}h_t$

```python
import torch
import torch.nn as nn

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.W_xh = nn.Linear(input_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size, bias=False)
        self.readout = nn.Linear(hidden_size, output_size)

    def forward(self, X, h=None):
        # X: (T, B, input_size), unrolled over time; h: (B, hidden_size), defaults to 0
        # readout is a separate map hidden_size -> output_size: for autoregressive
        # use, set output_size = input_size so the output Y_t can be fed back in
        # as X_{t+1}, distinct from the hidden state h that keeps recurring.
        T, B, _ = X.shape
        if h is None:
            h = X.new_zeros(B, self.hidden_size)
        outputs = []
        for t in range(T):
            h = torch.tanh(self.W_xh(X[t]) + self.W_hh(h))
            outputs.append(self.readout(h))
        return torch.stack(outputs), h
```

of a differential equation, can be approximated by a neural network with arbitrary precision and under moderate assumptions.

> THEOREM 8.1 (Universal dynamical system approximation)
> *Let D be an open subset of $\mathbb{R}^d$, $F : D \mapsto \mathbb{R}^d$ be a C^1 mapping, and $\tilde{K}$ a compact subset of D. Suppose that there is a subset $K \subset \tilde{K}$ such that any solution $x(t)$ with initial value $x(0) \in K$ of an ordinary differential equation*
>
> $$dx = F(x)dt,$$
>
> *that is defined on $0 \leq t < T < \infty$, and $x(t) \in K$ for any valid t. Then, for an arbitrary $\varepsilon > 0$, there exist an integer N and a continuous recurrent neural network with N neurons such that for an appropriate initial state of the network,*
>
> $$\max_{0 \leq t < T} |x(t) - u(t)| \leq \varepsilon,$$
>
> *with $u(t)$ the internal state of the network.*

Proof. The proof given by ichi Funahashi and Nakamura (1993) is too long for this textbook, but the scheme is as follows. There is an inequality in dynamical systems theory, Grönwall's inequality, that implies a lemma stating that if two vector fields F and $\tilde{F}$ are distant by ε, then their respective trajectories follow

$$|x(t) - \tilde{x}(t)| \leq \frac{\varepsilon}{L}(e^{Lt} - 1),$$

for all t, with L the Lipschitz constant of F.

In our case, t is bounded, and the right hand side is a constant multiplied by ε, relating the error on the trajectories to that of the vector fields, *i.e.* standard vector function approximation. We can make the field error arbitrarily small thanks to the standard universal approximation theorems for neural networks. ■

This has practical implication in control engineering because it means that RNN are as good as any other tool to perform system identification or adaptive inverse control, and yet instead of deriving the model from mathematical modeling, one can simply train the network on many samples.

8.2 LSTM

To prevent vanishing information from the hidden state, Hochreiter and Schmidhuber (1997) proposed a gating mechanism aimed at filtering which information gets into the hidden state and which gets out. The key principle is to learn when to memorize and when to forget, using a memory cell c_t for that purpose, called ***Long Short Term Memory*** or LSTM for short.

The information that we want to store into the cell is similar to that of a vanilla RNN,

$$\tilde{c_t} = \sigma_h(W_{c,x}x_t + W_{c,h}h_t),$$

and depends on the hidden state h_t. The memory cell itself is updated with

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c_t},$$

with i_t an input gate to let information in, and f_t a forget gate to reset the cell.

The input gate is given by

$$i_t = \sigma_g(W_{i,x}x_t + W_{h,x}h_t),$$

and the forget gate by

$$f_t = \sigma_g(W_{f,x}x_t + W_{h,x}h_t).$$

The output is the hidden state

$$h_t = o_t \odot \sigma_h(c_t),$$

computed thanks to an output gate

$$o_t = \sigma_g(W_{o,x}x_t + W_{o,h}h_t).$$

The activations σ_g are sigmoids, and σ_h is the hyperbolic tangent.

While LSTM do prevent the vanishing information problem, they are also notoriously hard to train. There are many sigmoids in the model, and any one of them saturating prevents proper backpropagation. There are also subject to another problem caused by the anti-causal link between the moment when the input is useful and the moment that the model decides to memorize the input. For example, in the sequence ***ADCCDBBCCADDB***, if the question is ***"when the sequence hits BB output the character next to the previous A"***, the answer is ***D***, but it requires to memorize ***D*** in the cell before the model sees the ***BB***. Thus, the input gate has to work in anticipation of what may be useful. This is a serious problem for all state based models, that is, models that encode a causal sequence into a hidden state, because the limited capacity of that state means that a choice has to be made on which input is retained. For some applications, ***e.g.***, anomaly detection, this is not a problem because important inputs are easily identifiable, but for some other tasks, like textual question answering, it is severely limiting.

8.2.1 GRU

The memory cell design of LSTM is cumbersome and can be simplified, as proposed by Cho et al. (2014) with the Gated Recurrent Unit, or GRU for short. GRU uses a reset gate

$$r_t = \sigma_g(W_{r,x}x_t + W_{r,h}h_{t-1}),$$

to compute a candidate hidden state

$$\tilde{h}_t = \sigma_h(W_{h,x}x_t + W_{h,h}(r_t \odot h_{t-1})),$$

almost like a vanilla RNN would do. Then the hidden state is updated by interpolating the old hidden state and the candidate

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t,$$

with z_t the update gate

$$z_t = \sigma_g(W_{z,x}x_t + W_{z,h}h_{t-1}).$$

This design has all the key concepts of LSTM, but fewer gates and is thus easier to train. We show a pytorch implementation of a GRU in Listing 8.2. As for the vanilla RNN, the forward is performed by unrolling the input sequence. In the case of autoregressive tasks, teacher forcing is used.

Listing 8.2 Gated Recurrent Unit as a PyTorch module: an update gate z_t and reset gate r_t control how much of the previous hidden state is kept versus overwritten by the candidate $\tilde{h}_t$, avoiding the vanishing-information problem of the vanilla RNN without a separate memory cell.

```
import torch
import torch.nn as nn

class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.W_zx = nn.Linear(input_size, hidden_size)
        self.W_zh = nn.Linear(hidden_size, hidden_size, bias=False)
        self.W_rx = nn.Linear(input_size, hidden_size)
        self.W_rh = nn.Linear(hidden_size, hidden_size, bias=False)
        self.W_hx = nn.Linear(input_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size, bias=False)
        self.readout = nn.Linear(hidden_size, output_size)

    def forward(self, X, h=None):
        # X: (T, B, input_size), unrolled over time; h: (B, hidden_size), defaults to 0
        T, B, _ = X.shape
        if h is None:
            h = X.new_zeros(B, self.hidden_size)
        outputs = []
        for t in range(T):
            z = torch.sigmoid(self.W_zx(X[t]) + self.W_zh(h))
            r = torch.sigmoid(self.W_rx(X[t]) + self.W_rh(h))
            h_tilde = torch.tanh(self.W_hx(X[t]) + self.W_hh(r * h))
            h = (1 - z) * h + z * h_tilde
            outputs.append(self.readout(h))
        return torch.stack(outputs), h
```

8.3 Transformers

To solve the problem of limited information in the hidden state, Vaswani et al. (2017) proposed to keep the entire sequence in memory and instead perform an online combination of all the past elements to perform the prediction. This is achieved through the use of ***attention***, defined as follows.

> Definition 8.2 (Attention)
> Given an input sequence X in matrix form, three projection matrices W_q, W_k and W_v projecting X to $\mathbb{R}^d$, ***Attention*** is defined as
> $$\sigma\left(QK^\top/\sqrt{d}\right)V,$$
> with $Q = XW_q$ called ***queries***, $K = XW_k$ called ***keys*** and $V = XW_v$ called ***values***. σ is the softmax operation.

Attention is often performed in a residual way,

$$Y = X + AV,$$

with $A = \sigma\left(QK^\top/\sqrt{d}\right)$ being denoted the ***attention matrix***.

The interpretation of attention is as follows: each token in the sequence is mapped to a query and a key, and the dot product between queries and keys encodes how much this key is relevant for that particular query. All pairs of query-key dot products are computed in the attention matrix and the softmax normalization is to ensure that its values are positive and sum to one, putting all keys in competition for each query. This normalized attention matrix is used as weights to combine the values, which are just a projection of the tokens in the sequence. The softmax thus ensures that the combination is convex and sparse, that is, that only a few tokens are being aggregated into the query.

If we imagine that the attention matrix is almost binary, maybe because only one key is very much aligned with the query and the softmax pushes all other values to zero, then for every token in the sequence, attention is selecting another token and merging it into the query. If we follow this strategy over L layers, then 2^L tokens will have been merged at the end of this process. Attention is thus very effective at combining the entire sequence with a shallow depth, even more so when the attention matrix is not almost binary.

In addition, attention does not store information in a hidden state, which means any previous information can be accessed by the current query. In our previous example with the long string, attention can solve this problem by first looking at the previous token to see if that's a B when the current token is a B, then the next layer puts attention on the closest A left of the current tokens, and finally the third layer puts attention on the tokens next to that A to fetch the answer D. Example of Attention based models learning that type of complex behavior using SGD only exist, for example in the estimation of an algorithm shown by Lepage et al. (2025).

Multi-head Attention. To allow the model to fetch information from more than one token, ***Multi-head Attention*** split the queries and the keys into multiple sub-vectors that are normalized independently using softmax. This models a case where several patterns of interest need to be combined, and each of the subspaces spanned by the query-key projections, called a ***head***, encodes one of them. The recombination of all heads is performed by concatenating them and adding a linear projection on top.

The listing for multi-head attention is shown in Listing 8.3. Modern implementations of multi-

Listing 8.3 Multi-head attention as a PyTorch module: queries, keys and values are split into `n_heads` independent d_{head}-dimensional slices, each running the attention of Definition 8.2 in parallel, before the concatenated heads are projected back to d_{model} and added residually.

```python
import torch
import torch.nn as nn

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0
        self.n_heads = n_heads
        self.d_head = d_model // n_heads
        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)
        self.W_o = nn.Linear(d_model, d_model, bias=False)

    def forward(self, X):
        # X: (B, T, d_model); each head attends independently on its own d_head slice
        B, T, _ = X.shape
        split = lambda Z: Z.view(B, T, self.n_heads, self.d_head).transpose(1, 2)
        Q, K, V = split(self.W_q(X)), split(self.W_k(X)), split(self.W_v(X))
        A = torch.softmax(Q @ K.transpose(-2, -1) / self.d_head ** 0.5, dim=-1)
        heads = (A @ V).transpose(1, 2).reshape(B, T, self.n_heads * self.d_head)
        return X + self.W_o(heads)
```

head attention are more effective and use low-level optimizations to fuse some operations. In addition, the attention matrix can take a lot of memory since it is quadratic with respect to the sequence length. Instead, it is more beneficial to recompute it during the backward pass than to store its activations as most modern GPU architectures are more often memory limited than computation limited.

Masked Attention. In the definition that we proposed in Definition 8.2, the causal aspect of the sequence is not taken into account, and indeed, tokens can attend other tokens in the future. This is solved by introducing a mask that set the attention to 0 for some tokens. In the case of causal sequences, the mask is a binary lower triangular matrix M and the attention is replaced with

$$A = \sigma(QK^\top/\sqrt{d}) \odot M,$$

with $\odot$ the element-wise product.

In fact, any mask can be used to prevent any token from attending any other tokens. For example, in the case of a graph, the adjacency matrix can be used as a mask to avoid nodes from combining information from nodes they are not connected to. Information from a node that is more than one hop away can still be combined by sequential aggregation through the layers.

Listing 8.4 A transformer encoder layer as a PyTorch module, built from Listing 8.3 and the LayerNorm of Listing 7.3: self-attention and the feed-forward MLP each get a residual connection followed by a LayerNorm, the post-LN convention from the original Transformer.

```
import torch.nn as nn

from attention import MultiHeadAttention
from layernorm import LayerNorm

class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, n_heads)
        self.norm1 = LayerNorm(d_model)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Linear(d_ff, d_model),
        )
        self.norm2 = LayerNorm(d_model)

    def forward(self, X):
        X = self.norm1(self.attn(X))    # attn already adds the residual (X + AV)
        X = self.norm2(X + self.ff(X))
        return X
```

8.3.1 Transformer architectures

In the original paper, Vaswani et al. (2017) proposed two distinct architectures to tackle natural language processing. The first architecture is called ***transformer encoder*** and processes all the tokens of the sequence with a combination of attention blocks and MLP,

$$X^{l+\frac{1}{2}} = X + AV,$$
$$X^{l+1} = X^{l+\frac{1}{2}} + \mathrm{FF}(X^{l+\frac{1}{2}}),$$

with FF an MLP, in a fully residual architecture.

In the transformer encoder, tokens of the sequences attend the other tokens of the sequence, and we call it ***self-attention***. We show a simple pytorch code for a transformer encoder layer in Listing 8.4.

The second architecture is a ***transformer decoder*** that takes two inputs, the input sequence X and a contextual sequence X_c that is used to provide information through the attention mechanism. The combination of attention and MLP is then

$$X^{l+\frac{1}{2}} = X + \sigma(QK_c^\top/\sqrt{d})V_c,$$
$$X^{l+1} = X^{l+\frac{1}{2}} + \mathrm{FF}(X^{l+\frac{1}{2}}),$$

with $K_c = X_c W_k$ and $V_c = X_c W_v$ the keys and values of the context sequence X_c.

In that architecture, the tokens of X can only attend the tokens of X_c and we call it *cross-attention*. Self-attention and cross-attention can be modeled with masking, although it is more computationally efficient to avoid computing values that are to be zeroed by the mask.

The main benefit of transformers is that they do not require the unrolling that RNNs do. All the tokens can be processed in parallel, with causality respected thanks to masking, which means that increased computational resources lead to faster training, even for very long sequences.

Universal approximation theorem. Transformers are universal sequence-to-sequence approximators, as shown by Yun et al. (2020) in the following theorem.

> Theorem 8.3 (Universal approximation for Transformers)
> *Define the distance d_p as*
>
> $$d_p(f, g) = \left(\int \|f(X) - g(X)\|_p^p dX \right)^{1/p}.$$
>
> *Let $1 \leq p \leq \infty$ and $\epsilon > 0$, then for any given $f \in \mathcal{F}$ the set of continuous functions that map a compact domain in $\mathbb{R}^{d \times n}$ to $\mathbb{R}^{d \times n}$, there exists a transformer g with learned positional encoding such that $d_p(f, g) \leq \epsilon$.*

Proof. The proof relies on a property of transformers called *contextual mapping* as expressed by the following lemma.

> Lemma 8.4 (Contextual mapping (informal))
> *Attention is a contextual mapping q on $\mathbb{R}^{d \times n}$, that is:*
>
> - *For any $X \in \mathbb{R}^{d \times n}$ with different entries, $q(X)$ has different entries.*
> - *For any $X, X' \in \mathbb{R}^{d \times n}$ that differ at least by one element, then all entries of $q(X)$ and $q(X')$ are different.*

Since the attention part of transformer is a contextual mapping, it can map a token to a number that is unique to both the token and the rest of the sequence. The MLP that follows in the transformer can map those unique values to that of the required output thanks to the universal approximation theorems for MLPs. ■

8.3.2 Fast alternative to attention

There have been several attempts to accelerate attention and most notably to avoid paying the quadratic cost. Reformer by Kitaev et al. (2020) reduce the attention matrix to approximate neighbors only that can be found efficiently, with the assumption that the attention matrix is very sparse and close enough to 0 for tokens that are not in the neighbor set. Linformer by Wang et al. (2020) proposes to compute a low-rank approximation of the kernel matrix, but this fixes the size of the sequence.

In this section, we show an alternative that we proposed in (Picard et al., 2026) that is based on polynomial kernels called PoM for Polynomial Mixer. We separate the stream in two branches. The first branch is a sequence representation that aggregates information about the sequence X into a

state representation $H(X)$ such that

$$H(X) = \sum \left[\sum_{p=1}^{k} \alpha_p \odot h(W_h X)^p \right].$$

The second branch consists in a gate $\sigma(W_s X)$ that selects information from H resulting in the following operation

$$\text{PoM}(X) = W_o \left[\sigma(W_s X) \odot H(X) \right].$$

We showed that PoM has the same contextual mapping properties as attention and that a transformer build with it instead of attention is a universal approximator. However, the cost of PoM is linear with respect to the sequence size, which makes it appealing like RNNs for causal sequences. The trade-off is that like RNNs, PoM has a finite hidden space and thus cannot store an infinite amount of information in it. The capacity is in theory limited by the dimension of the hidden space and the degree of the polynomial.

8.4 Applications

Transformers have had a tremendous impact on the entire field of machine learning by removing the inductive bias of convolutional architectures that assume locality is a relevant aspect. In this section, we show two applications of transformers, to natural language processing which gave rise to the Large Language Models powering the Generative AI revolution, and to image processing with the visual transformer that are the ***de facto*** standard architecture for extracting image features.

8.4.1 Language modeling

Transformers are a perfect tool to perform language modeling since co-occurrences of words in a specific order, but not necessarily locally, is a key aspect of human languages as theorized by the distributional hypothesis introduced by Harris (1954). To process sequences of words, the first step is to tokenize them, ***i.e.***, to enumerate a set of combination of characters called a dictionary such that any text can be represented by a sequence of indices in the dictionary. Tokenization is often based on information theory, for example in byte-pair encoding by Sennrich et al. (2016) that merges the most frequent pairs of adjacent bytes to build the dictionary. Each token can then be transformed from an index to a vector via an embedding matrix by either outputting the matrix row corresponding to the index, or more formally by multiplying the one-hot encoding of the index by the embedding matrix.

The transformer then processes the set of vectorized tokens, depending on the task at hand. To learn text representations, masked modelling like BERT has been propose by Devlin et al. (2019) and consists in masking some of the words in the sentence and use attention to recover them. This is similar in idea to the early word representations proposed by Mikolov et al. (2013) only applied to the entire sequence. BERT and its followers provide much stronger text features than handcrafted descriptors.

The real breakthrough has been the introduction of GPT for ***Generative Pretrained Transformer*** by Radford et al. (2018). GPT is a simple transformer architecture that is trained with a causal mask solely on next token prediction. It turns out that the scaling laws of such model are very robust, and as such GPTs have been scaled to trillions of parameters trained on trillions of tokens.

To train the largest models without enduring the extreme cost, a ***Mixture of Experts*** (MoE) strategy is used. After the attention, there are several MLP, and each token is routed through only one

of them with a small router (itself a small MLP outputting a softmax distribution over the MLPs). This MoE strategy increases the total parameter count while keeping the number of ***active*** parameters the same since the tokens are processed by only one of the MLPs.

We provide in Listing 8.5 the code for a small GPT in pytorch. Unfortunately, such model cannot be trained in a way that obtains meaningful results without professional hardware. Limiting the training set to only 100 billions of tokens, a small model (less than 1B parameters) cannot output something more interesting than simple text completion, certainly nothing comparable to the reasoning that state-of-the-art models are now capable of.

8.4.2 Image modeling

Another application domain that has been changed by transformers is image processing. Removing the convolutional inductive bias from neural networks seems counter-intuitive at first because images have this locality property (objects are connected components in our universe) and also linear filters (a.k.a. convolution) have been the landmark of image processing for as long as the field exists.

Nonetheless, Dosovitskiy et al. (2021) showed that training a transformer on image by converting small patches into tokens and learning a 2-dimensional positional encoding, scales more than convolutional networks. Such architecture is called a ***Vision Transformer*** or ViT for short. It seems that removing the inductive bias from the design of the architecture lets the model learn the equivalent properties from the data and that this is a way to better exploit massive amounts of data. In their work, the authors of ViT showed that they could surpass the performances of all other architectures by training the transformer on a massive dataset containing 300 million images and then fine-tuning it on the target task. Since then, unsupervised pretraining of ViT, such as what has been proposed by Caron et al. (2021), has become the standard procedure to obtain image features.

Not long after ViT has been introduced, Radford et al. (2021) proposed to use it to learn a cross-modal embedding between text and images. The method, called CLIP for ***Contrastive Language–Image Pre-training***, consists in aligning the representation from a text transformer to that of an image transformer using a massive dataset of image-text pairs crawled from the internet. CLIP has been extremely successful at providing representation for text and images that can be combined.

We show in Listing 8.6 a pytorch implementation of a ViT. We trained this architecture on our Bluebell dataset, using heavy data-augmentation (random flip of the images and random quarter turn rotations) and dropout to avoid overfitting since this is such a small dataset. Nonetheless, we obtained an average training accuracy of 99.2% across the different splits, and an average validation accuracy of 95.7%. This is the best result so far, with limited overfitting as a bonus. We could probably gain a few more percent by increasing the size of the architecture and relying on even heavier augmentations and regularization, but given the size of the dataset 4.3% error corresponds only to a few mislabeled images and we are reaching the limit of what is meaningfully measurable.

8.5 Exercises

Exercise 8.1 — Impossible sequence for RNN. Construct a periodic sequence that a vanilla RNN with a 1-dimensional hidden state provably cannot represent, but a 2-dimensional hidden state can.

Exercise 8.2 — Copy task with an RNN and a GRU. Train an RNN listing on a synthetic copy task and show where it fails on long-range dependencies compared to a GRU.

Listing 8.5 A small GPT-style decoder-only Transformer as a PyTorch module: each block takes a causal mask (an upper-triangular boolean mask passed to `nn.MultiheadAttention`) so position t can only attend to positions $\leq t$. The head is applied at every position, producing next-token logits everywhere in parallel for teacher-forced training.

```
import torch
import torch.nn as nn

class CausalTransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.attn = nn.MultiheadAttention(d_model, n_heads, batch_first=True)
        self.norm1 = nn.LayerNorm(d_model)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Linear(d_ff, d_model),
        )
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, causal_mask):
        attn_out, _ = self.attn(x, x, x, attn_mask=causal_mask, need_weights=False)
        x = self.norm1(x + attn_out)
        x = self.norm2(x + self.ff(x))
        return x

class SmallGPT(nn.Module):
    def __init__(self, vocab_size, max_len=256, d_model=192, n_heads=4, n_layers=6,
        d_ff=768):
        super().__init__()
        self.token_embed = nn.Embedding(vocab_size, d_model)
        self.pos_embed = nn.Parameter(torch.zeros(1, max_len, d_model))
        self.blocks = nn.ModuleList([CausalTransformerBlock(d_model, n_heads, d_ff)
            for _ in range(n_layers)])
        self.norm = nn.LayerNorm(d_model)
        self.head = nn.Linear(d_model, vocab_size)

    def forward(self, tokens):
        T = tokens.shape[1]
        x = self.token_embed(tokens) + self.pos_embed[:, :T]
        causal_mask = torch.triu(torch.ones(T, T, dtype=torch.bool,
            device=tokens.device), diagonal=1)
        for block in self.blocks:
            x = block(x, causal_mask)
        x = self.norm(x)
        return self.head(x)
```

Listing 8.6 A small Vision Transformer as a PyTorch module: a strided convolution patchifies and embeds the image in one step and a stack of standard transformer blocks (`nn.MultiheadAttention` + MLP, each with a residual connection and LayerNorm). Since attention is permutation-invariant, a learned position embedding is added to the patch sequence, and a learned `cls_token` is prepended and read off by the classifier at the end.

```
import torch
import torch.nn as nn

class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.attn = nn.MultiheadAttention(d_model, n_heads, batch_first=True)
        self.norm1 = nn.LayerNorm(d_model)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Linear(d_ff, d_model),
        )
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x):
        attn_out, _ = self.attn(x, x, x, need_weights=False)
        x = self.norm1(x + attn_out)
        x = self.norm2(x + self.ff(x))
        return x

class SmallViT(nn.Module):
    def __init__(self, n_classes, img_size=64, patch_size=8, in_channels=3,
                 d_model=192, n_heads=4, n_layers=6, d_ff=768):
        super().__init__()
        n_patches = (img_size // patch_size) ** 2
        self.patch_embed = nn.Conv2d(in_channels, d_model, kernel_size=patch_size,
            stride=patch_size)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, d_model))
        self.pos_embed = nn.Parameter(torch.zeros(1, n_patches + 1, d_model))
        self.blocks = nn.Sequential(*[TransformerBlock(d_model, n_heads, d_ff) for _
            in range(n_layers)])
        self.norm = nn.LayerNorm(d_model)
        self.classifier = nn.Linear(d_model, n_classes)

    def forward(self, x):
        x = self.patch_embed(x).flatten(2).transpose(1, 2)
        cls = self.cls_token.expand(x.shape[0], -1, -1)
        x = torch.cat([cls, x], dim=1) + self.pos_embed
        x = self.blocks(x)
        x = self.norm(x)
        return self.classifier(x[:, 0])
```

Exercise 8.3 — Causal mask for attention. Add a causal mask to `attention.py` and empirically verify that a token's output is unaffected by changes to future tokens.

Exercise 8.4 — Computational cost of attention. Compute the forward pass cost of attention in FLOP and verify it is indeed quadratic in the sequence length.

Bibliography

Athey, S. and Imbens, G. (2016). Recursive partitioning for heterogeneous causal effects. ***Proceedings of the National Academy of Sciences***, 113(27):7353–7360.

Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2019). Reconciling modern machine-learning practice and the classical bias–variance trade-off. ***Proceedings of the National Academy of Sciences***, 116(32):15849–15854.

Bordes, A., Bottou, L., and Gallinari, P. (2009). Sgd-qn: Careful quasi-newton stochastic gradient descent. ***Journal of Machine Learning Research***, 10:1737–1754.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In ***Proceedings of the fifth annual workshop on Computational learning theory***, pages 144–152.

Bousquet, O. and Elisseeff, A. (2002). Stability and generalization. ***Journal of machine learning research***, 2(Mar):499–526.

Breiman, L. (2001). Random forests. ***Machine learning***, 45:5–32.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). Classification and regression trees.

Caron, M., Touvron, H., Misra, I., Jégou, H., Mairal, J., Bojanowski, P., and Joulin, A. (2021). Emerging properties in self-supervised vision transformers. In ***2021 IEEE/CVF international conference on computer vision (ICCV)***, pages 9630–9640. IEEE.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In ***Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining***, pages 785–794.

Cho, K., Van Merriënboer, B., Gulçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder–decoder for statistical machine

translation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1724–1734.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20:273–297.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

Défossez, A., Bottou, L., Bach, F., and Usunier, N. (2022). A simple convergence proof of adam and adagrad. *Transactions on Machine Learning Research*.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*.

Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.

Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, pages 148–156.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10(5):447–474.

Harris, Z. S. (1954). Distributional structure. *Word*, 10(2-3):146–162.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hendrycks, D. and Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

ichi Funahashi, K. and Nakamura, Y. (1993). Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801–806.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr.

Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR (Poster)*.

Kitaev, N., Kaiser, L., and Levskaya, A. (2020). ReFormer: The efficient transformer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

LeCun, Y. (1988). A theoretical framework for back-propagation. In *Proceedings of the 1988 Connectionist Models Summer School*, pages 21–28.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lepage, S., Mary, J., and Picard, D. (2025). Markov chain estimation with in-context learning. In *Gretsi*.

Maas, A. L., Hannun, A. Y., Ng, A. Y., et al. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA.

Malach, E., Yehudai, G., Shalev-Schwartz, S., and Shamir, O. (2020). Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In *ICLR Workshop*.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

Pelillo, M. (2014). Alhazen and the nearest neighbor rule. *Pattern Recognition Letters*, 38:34–37.

Picard, D., Dufour, N., Degeorge, L., Ghosh, A., Allegro, D., Ravaud, T., Perron, Y., Sautier, C., Baltaci, Z. S., Meng, F., Kalleli, S., López-Rauhut, M., Loiseau, T., Albouy, S., Baena, R., Vincent, E., and Landrieu, L. (2026). Pom: A linear-time replacement for attention with the polynomial mixer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Findings*, pages 2544–2553.

Platt, J. (1998). Sequential minimal optimization: A fast algorithm for training support vector machines.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. (2021). Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.

Raginsky, M. and Recht, B. (2026). Separating geometry from probability in the analysis of generalization. *arXiv preprint arXiv:2604.19560*.

Rahimi, A. and Recht, B. (2007). Random features for large-scale kernel machines. In *Advances in neural information processing systems*.

Rakotomamonjy, A., Canu, S., and Smola, A. (2005). Frames, reproducing kernels, regularization and learning. *Journal of Machine Learning Research*, 6(9).

Reddi, S. J., Kale, S., and Kumar, S. (2018). On the convergence of adam and beyond. In *International Conference on Learning Representations*.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Roux, N., Schmidt, M., and Bach, F. (2012). A stochastic gradient method with an exponential convergence _rate for finite training sets. *Advances in neural information processing systems*, 25.

Schölkopf, B. and Smola, A. J. (2002). *Learning with kernels: support vector machines, regularization, optimization, and beyond*.

Sennrich, R., Haddow, B., and Birch, A. (2016). Neural machine translation of rare words with subword units. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 1715–1725.

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

Shalev-Shwartz, S., Singer, Y., and Srebro, N. (2007). Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of the 24th international conference on Machine learning*, pages 807–814.

Shalev-Shwartz, S. and Zhang, T. (2013). Stochastic dual coordinate ascent methods for regularized loss minimization. *Journal of Machine Learning Research*, 14(1).

Siegelmann, H. T. and Sontag, E. D. (1992). On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *ICLR*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Telgarsky, M. (2016). Benefits of depth in neural networks. In *Conference on learning theory*, pages 1517–1539. PMLR.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

Vapnik, V. N. (1995). *The nature of statistical learning theory*. Springer-Verlag.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. (2020). Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*.

Williams, C. and Seeger, M. (2000). Using the nyström method to speed up kernel machines. *Advances in neural information processing systems*, 13.

Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S., and Kumar, S. (2020). Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*.