Introduction to Machine Learning

Flavio Schneider and Soel Micheletti

Spring 2020

Table of Symbols

Symbol or Acronym	Meaning
$a, b, c, \alpha, \beta, \gamma$	Scalars (lowercase).
$\mathbf{x}, \mathbf{y}, \mathbf{z}, \boldsymbol{\alpha}$	Vectors (bold lowercase).
x_i	Index at i of vector $\mathbf{x}$.
$\mathbf{A}, \mathbf{B}, \mathbf{C}$	Matrices (bold uppercase).
$\mathbf{A}_{i,:}$	Row i of matrix $\mathbf{A}$ as row vector.
$\mathbf{A}_{:,j}$	Column j of matrix $\mathbf{A}$ as column vector.
$\mathbf{a}_i$	Row or column i (vector) of matrix $\mathbf{A}$ (depends on context).
$a_{i,j}$	Index at row i and column j of matrix $\mathbf{A}$.
(a, b, c)	Row vector (ordered touple) of a, b, c.
$[a, b, c]$	Column vector of a, b, c.
$(\mathbf{x}, \mathbf{y}, \mathbf{z})$	Matrix with $\mathbf{x}, \mathbf{y}, \mathbf{z}$ as columns.
$[\mathbf{x}, \mathbf{y}, \mathbf{z}]$	Matrix with $\mathbf{x}, \mathbf{y}, \mathbf{z}$ as rows.
$\{x, y, z\}$	Set of elements (unordered).
$\mathbf{x}^\top, \mathbf{A}^\top$	Transpose of vector or matrix.
X,Y,Z	Random variables (sans-serif uppercase).
$\boldsymbol{\mathsf{X}}$,$\boldsymbol{\mathsf{Y}}$,$\boldsymbol{\mathsf{Z}}$	Multivariate random variables (sans-serif bold uppercase).
$\mathbb{P}[\mathsf{X} = x]$	Probability that the random variable X is realized as x.
$\mathbb{E}[\mathsf{X}]$	Expected value of X.
$\mathbb{V}[\mathsf{X}]$	Variance of X.
$\forall x$	Universal quantifier: for all x
$\exists x$	Existential quantifier: there exist x
$a \in A$	The set A contains the element a.
$\varnothing$	Empty set.
$a \coloneqq b$	The value a is defined as b.
$a \eqqcolon b$	The value b is defined as a.
$\mathbb{N}$	The set of natural numbers: $\{1, 2, 3, \dots\}$.
$\mathbb{N}_0$	The set of natural numbers with 0: $\{0, 1, 2, \dots\}$.
$\mathbb{Z}$	The set of integers.
$\mathbb{R}$	The set of real numbers.
$\mathbb{C}$	The set of complex numbers.
$[n]$	All integers from 1 to n: $[n] \coloneqq \{1, \dots, n\}$.
i.i.d.	Independent and identically distributed.
i.e.	That is... / This means...
e.g.	For example...
s.t.	Such that...
etc.	And so on.
et al.	And others.

Contents

Table of Symbols iii

Contents v

1 Introduction 1
- 1.1 Supervised learning . . . 2
- 1.2 Unsupervised learning . . . 4

Supervised Learning 5

2 Regression 6
- 2.1 Linear regression . . . 6
- 2.2 Polynomial Regression . . . 8
- 2.3 Prediction Error . . . 8
- 2.4 Cross Validation . . . 11
- 2.5 Model Selection . . . 12
- 2.6 Regularization . . . 12
- 2.7 Standardization . . . 13

3 Classification 15
- 3.1 Binary Classification . . . 15
- 3.2 Perceptron Algorithm . . . 16
- 3.3 Stochastic Gradient Descent . . . 17
- 3.4 Support Vector Machine . . . 17
- 3.5 Feature Selection . . . 18
- 3.6 Class Imbalance . . . 21
- 3.7 Multi-class Classification . . . 24

4 Kernels 27
- 4.1 Feature Explosion Problem . . . 27
- 4.2 Polynomial Kernels . . . 29
- 4.3 Kernelized Perceptron . . . 30
- 4.4 Kernel Properties . . . 31
- 4.5 Infinite Dimensional Kernels . . . 32
- 4.6 Kernelized SVM . . . 34
- 4.7 Kernelized Linear Regression . . . 34

5 Neural Networks 36
- 5.1 Introduction . . . 36
- 5.2 General Neural Network . . . 37
- 5.3 Forward Propagation . . . 39
- 5.4 Objective . . . 40
- 5.5 Computational Graphs . . . 41
- 5.6 Back-Propagation . . . 45
- 5.7 Weight Initialization . . . 48
- 5.8 Optimizers . . . 50
- 5.9 Overfitting . . . 50

5.10 Regularization 51
5.11 Convolutional Neural Networks (CNNs) 53
5.12 Other 58

6 Probabilistic Approach to Supervised Learning **60**
6.1 Bias Variance Tradeoff 62
6.2 Logistic Regression 64
6.3 Bayesian Decision Theory 66
6.4 Generative Modeling 68

Unsupervised Learning **72**

7 Classification **73**
7.1 Clustering 73
7.2 K-Means Clustering 74

8 Regression **77**
8.1 Dimension Reduction 77
8.2 Principal Component Analysis (PCA) 77
8.3 Kernel PCA 82
8.4 Autoencoders 84

9 Probabilistic Approach to Unsupervised Learning **87**
9.1 Mixture Distribution 87
9.2 Gaussian Mixtures Model 88

Introduction 1

1.1 Supervised learning 2
1.2 Unsupervised learning 4

Example 1.0.1 (Spam E-Mail) Imagine that one has to write a program that, given an E-Mail message, decides whether the E-Mail is spam or not. In order to solve the task one can decide a set of rules and classify the E-Mail accordingly. An example of rule could be: *if the text body contains "login here", classify it as spam*. This approach seems infeasible because it may be easy to hack and choosing a good set of rules seems very difficult. Here is where machine learning comes into play: we want an *automatic discovery of rules from training data*. Concretely one collects a large amount of data (i.e. examples of spam/ non-spam E-Mails) and then uses general purpose learning methods to discover a decision rule that (hopefully) generalizes well to new examples which are not part of the original training data.

Now we discuss a very broad definition of Machine Learning given by Tom Mitchell and we mention a couple of observations which show how design choices affect the algorithm that one will use in the end:

> A computer is said to learn from experience E with respect to some task T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

- A straightforward example of performance measure could be *accuracy* (i.e. the number of mistakes out of number of decisions made on examples which are not part of the training set). Although it might seem a natural choice, this is not always what one wants. In the case of Example 1.0.1 a false positive (i.e. a legitimate E-Mail classified as spam) might be considered worse than a false negative (i.e. a spam E-Mail classified as non-spam).
- In Example 1.0.1 we discussed a *binary classification* task. However in the E-Mail classification problem this might be not exactly what the designer wants. Concretely, it could be better to have three different labels: spam (which are put in the junk box), non-spam (which are put in the inbox) and doubt (which are put in the inbox but marked with a warning sign).

Nowadays, Machine Learning is a very hot topic in the news. In facts, both Machine Learning and Artificial Intelligence play a very important role in today's society. This happens because we are in the era of *big data*, i.e. we are in an era in which a lot of data (about human behaviour and other phenomena) are available. If analysed properly, one can get value out of this data, and Machine Learning plays a core role in this value chain.

Machine Learning algorithms are usually divided in two major categories: *supervised learning* and *unsupervised learning*. In this chapter we see a brief overview of both types of learning, which will be the core of this script.

1.1 Supervised learning

Supervised learning can be though of as *learn from labeled examples*. In facts, the goal of supervised learning is learning functions of the form

$$f : X \to Y$$

Concretely, we want to learn a mapping from inputs in the set X to outputs in the set Y. In the case of Example 1.0.1, X is the set of all possible E-Mail messages and Y is the label of the input message (i.e. spam/ not-spam).

We can categorize supervised learning in the following subcategories:

- **Classification:** where Y is a discrete set of labels. When the cardinality of Y is two, we have *binary classification*, otherwise *multi-label classification*. A canonical example of multi-label classification is *ImageNet*, a dataset of several images used for object identification (i.e. given an image, one has to decide which of 1000 possible objects is the main one in the picture). In this case, X is the vector representing the pixels of the image and Y is a vector of dimension 1000 where each component represents the probability that the corresponding object is the main one in the image.
- **Regression:** where Y is a real (continuous) value, often in form of a vector. Some examples of regression are given in the following table:

X	Y
Flight route	Delay
Real estate objects	Price
Patient and Drug	Effectiveness
User and articles	Products to display
Image	Sentence in natural language
Text in English	Text in Italian
Ugly JS Code	Nice JS Code

Table 1.1: Examples of regression.

In order to understand how the general supervised learning pipeline works we introduce some definitions.

Definition 1.1.1 (Labeled Dataset) *A labeled dataset D is a set of m tuples. Formally $D := \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$. We refer to $\mathbf{x}_i \in \mathbb{R}^n$ as the feature vector of the i^{th} element.*

Definition 1.1.2 (Training Data) *The training data $T \subseteq D$ is a subset of the labeled dataset that the learning algorithm will use to train the model.*

Definition 1.1.3 (Test Data) *The test data $T' \subseteq D$ is a subset of the labeled dataset (usually s.t. $T \cap T' = \varnothing$) used to evaluate the performance of the model.*

In general (and at a high level) the supervised learning pipeline looks as follows:

1. Gather some *labeled dataset* (i.e. examples of elements in X associated to the right value in Y).
2. Pick some *training data* and some *test data* from the labeled dataset.
3. Use some learning algorithm on the training data in order to find a (hopefully good) mapping from X to Y.
4. Use the found mapping on the test data to evaluate its performance according to some metrics. If the model works well one can use it to predict the output from real world data.

Now we discuss some crucial aspects of this pipeline:

- **Representation of data:** since we use general-purpose learning algorithms we have to choose a representation that works for different kinds of inputs (e.g. both on text and images). In this course we will mostly use *feature vectors*, i.e. every $x \in X$ will be represented as a vector in $\mathbb{R}^d$, where d is the number of features we consider (and must be chosen carefully). An example of data representation for words is called *bag-of-words*: we suppose that a language has at most $d = 100000$ words and we represent each document as vector x in $\mathbb{R}^d$, where x_i counts the occurrences of word i in the document. This is very simple, but it has some drawbacks: some words are more important than others (e.g. the word *no* can be more informative than the word *the*), this scheme ignores the order of the words in the sentence (i.e. the mapping between documents and vectors is not injective), it is dependent on the length of the document and it ignores the semantics of the language. All these drawbacks can be mitigated with more involved versions of the same concept.
- **Model fitting:** given training examples (i.e. feature vectors with associated labels), it is necessary to find a *decision rule* to learn the desired function. Examples of decision rules are hyperplanes, linear decision trees, random forests, and deep neural networks.
- **Prediction and generalization:** at this point we have a model that, given an element of X, returns an element of Y. This model has two desirable properties: *goodness of fit* (i.e. it must have a good performance on the test set) and *complexity* (i.e. it should not be too complex). In general when the model is too simple we speak of *underfitting* and when the model is too complex we speak of *overfitting*. Ideal models are simultaneously statistically and computationally efficient.

In order to understand the concepts above, it is useful to apply them concretely on Example 1.0.1. We can represent E-Mails as bag-of-words (optimized in the best possible way) and represent each E-Mail of the training set in a $d-$dimensional vector space. Then one has to find a decision rule, e.g. a partition of the vector space in regions R and R' such that:

- $R \cap R' = \varnothing$
- $R \cup R' = \mathbb{R}^d$
- The spam E-Mails of the training set are in R
- The not-spam E-Mails of the training set are in R'

We can use this model (which, since we are in the case of classification is called *classifier*) to classify a new E-Mail as follows: we represent the new E-Mail in bag-of-words and we check, whether this vector is in R or R'. If the model is good enough we have a high probability of having done the right choice. Keep in mind that we aim to have *goodness of fit* and *reasonable complexity of the regions* at the same time.

1.2 Unsupervised learning

The other very famous class of Machine Learning algorithms is known as *unsupervised learning*. This means learning without supervision, i.e. the dataset does not have any labels.

Definition 1.2.1 (Unlabeled Dataset) *An unlabeled dataset D is a set of m vectors. Formally $D := \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. We refer to $\mathbf{x}_i \in \mathbb{R}^n$ as the feature vector of the i^{th} element. The difference to the labeled dataset is that we don't have any label.*

In some sense we are still trying to learn the same function f we introduced for supervised learning and the steps of the pipeline are essentially the same (i.e. training data, learning algorithm, model, prediction on test data). Two canonical classifications of unsupervised learning algorithms are:

- **Clustering:** can be thought of as *unsupervised classification*. Here we have a set of data without labels as input and we want to assign each vector input to a *cluster* (i.e. a group of similar data points) in order to infer the label *a posteriori*.
- **Dimension reduction:** can be thought of as *unsupervised regression*. Here we want to find a lower dimension of the dataset (which maybe can even be visualized) in order to have more efficient computation. The goal of dimension reduction is preserving as many features as possible, otherwise having the data in a lower-dimensional space would be useless.

Common goals in unsupervised learning algorithms are finding good data representation (a form of compression). The objective, however, is often not as clear as in supervised learning tasks. Examples of applications where an unsupervised learning approach was used are face recognition, anomaly detection, images generation, network inference, and many more.

Supervised Learning

Regression 2

2.1 Linear regression 6
2.2 Polynomial Regression 8
2.3 Prediction Error 8
2.4 Cross Validation 11
2.5 Model Selection 12
2.6 Regularization 12
2.7 Standardization 13

2.1 Linear regression

In its most general form, regression has the goal to learn a function f of the form:

$$f : \mathbb{R}^d \to \mathbb{R}$$

We have to ask ourselves two fundamental questions:

1. What type of functions should we consider?
2. How should we measure the goodness of fit?

In this section we talk about *linear regression*, but as we will see later, the same ideas apply also to other types of regression. Linear regression is of the form $y \approx f(x)$, where f is a linear function, i.e. a function that can be written as:

$$f(x) = w_1 x_1 + \cdots + w_d x_d + w_0 = \mathbf{w}^T x + w_0$$

where $\mathbf{w} = [w_1, \ldots, w_d]$ and $\mathbf{x} = [x_1, \ldots, x_d]$. Without loss of generality, we can use homogeneous coordinates and write:

$$f(x) = \mathbf{w}^T x$$

with $\mathbf{w} = [w_1, \ldots, w_d, w_0]$ and $\mathbf{x} = [x_1, \ldots, x_d, 1]$.

How do we quantify goodness of fit? There are multiple possible design choices (and, depending on which one we peek, we will use different algorithms). In this case we minimize the squared sum of residuals. Concretely, given the test data $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ we define the residual r_i as $y_i - f(\mathbf{x}_i) = y_i - \mathbf{w}^T \mathbf{x}_i$. At the end we want to minimize $\hat{R}(\mathbf{w}) = \sum_{i=1}^n r_i^2 = \sum_{i=1}^n \left(y_i - \mathbf{w}^T \mathbf{x}_i\right)^2$.

Now the question is: given a dataset $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, how do we find the optimal vector? Formally we want to find $\hat{\mathbf{w}}$ defined as:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \left(y_i - \mathbf{w}^T \mathbf{x}_i\right)^2$$

In this particular case, we have a closed-form solution. In facts, we can write the optimization problem as an (overdetermined) system of equations:

$$\underbrace{\begin{bmatrix} x_{1,1} & \dots & x_{1,d} & 1 \\ x_{2,1} & \dots & x_{2,d} & 1 \\ & \dots & & \\ x_{n,1} & \dots & x_{n,d} & 1 \end{bmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \\ w_0 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}}$$

And the least square solution gives us that the optimal weights are given by:

$$\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T y$$

However, most of the problems we solve in Machine Learning don't have a closed form solution. Also in this particular case, it might not be always necessary to use the closed-form solution since this is less efficient (you need to do matrix multiplication and solve a linear system of equations) and is not always necessary to get an optimal solution (an arbitrarily good approximation is always good enough).

In general a widely used algorithm to solve optimization problems is called *gradient descent*. The algorithm is very simple and looks as follows:

Algorithm 2.1: Gradient descent

1 Start at an arbitrary $\mathbf{w_0} \in \mathbb{R}^d$
2 ***for*** $t = 0, 1, 2, \dots$
3 $\quad \mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla \cdot \hat{R}(\mathbf{w}_t)$

where η_t is called *learning rate*. If the learning rate is chosen properly (in this case a learning rate of 0.5 would work), the algorithm converges to the optimum on convex functions*, i.e. on functions such that:

$$\forall x, x', \lambda \in [0,1] \text{ it holds that } f\left(\lambda x + (1-\lambda)x'\right) \leq \lambda f(x) + (1-\lambda) f(x')$$

Here we squared the residuals, but there are many other possible (and meaningful) loss functions. In general, the choice of the loss function introduces trade-offs:

- $|r|$: we have zero error with the right weights; no closed solution; is less sensitive to noise than the square loss.

* This condition is sufficient but non necessary. For non-convex functions this method sometimes converges to an optimum and sometimes to a stationary point.

- $|r|^p$ (for $p > 1$): convex function; error almost zero for errors which are less than one in absolute value but very sensitive to noise otherwise; might be useful if we want that all points are equally important.
- $|r|^p$ (for $p < 1$): not convex, but still possible to use gradient descent because of the shape of the function; robust to noise

2.2 Polynomial Regression

Often fitting a linear model doesn't work well because we underfit the data, thus instead we will use a polynomial.

Goal

Given a set of feature vectors $\mathbf{x_1}, \dots, \mathbf{x_n}$ where $\mathbf{x_i} \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$), and a set labels $y_1, \dots, y_n$ where $y_i \in \mathbb{R}$ (which can be represented as a vector $\mathbf{y} \in \mathbb{R}^n$).

Output the coefficient vector $\mathbf{w} \in \mathbb{R}^d$ such that

$$f(\mathbf{x_i}) := \sum_{j=1}^{D} w_j \phi_j(\mathbf{x_i}) \approx y_i, \quad \forall i \in \{1, \dots, n\} \tag{2.1}$$

where $\phi(\mathbf{x_i}) :=$ vector of all monomials of degree up to n in $x_{i,1}, \dots, x_{i,d}$, where $D = |\phi(\mathbf{x})|$. [a]

[a] In 2-d we have $\phi(\mathbf{x_i}) = \tilde{\mathbf{x}}_\mathbf{i} = [1, x_{i,1}, x_{i,2}, x_{i,1}^2, x_{i,2}^2, x_{i,1} \cdot x_{i,2}, \dots]^T$

Notice that to find the coefficient vector $\mathbf{w}$ we can still use linear regression, to do so we compute the values of the function ϕ as a new vectors $\tilde{\mathbf{x}}_\mathbf{i} = \phi(\mathbf{x_i})$, and then solve the problem using some standard linear regression method:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \mathbf{w}^T \tilde{\mathbf{x}}_\mathbf{i}\right)^2 \tag{2.2}$$

2.3 Prediction Error

Choosing the degree of the polynomial is a critical task. If we use a high degree polynomial on noisy data we fit the error, and hence the prediction error will increase. This situation is known as *overfitting*. Conversely, if we use a polynomial with (too) low degree, we don't have a model that is powerful enough to properly fit the data. This situation is known as *underfitting*. An important task in regression is choosing a model that is powerful enough (in order to avoid underfitting), but also not too complex (in order to avoid overfitting). The prediction error is a useful tool to find the best degree which optimizes the goodness of fit.

We start by assuming that each touple in the training dataset, is generated *i.i.d.* from some unknown distribution P:

$$(\mathbf{x_i}, y_i) \sim P(\mathbf{X}, \mathbf{y}) \tag{2.3}$$

then we can define the notion of error.

Definition 2.3.1 (Expected Error) *The expected error (or true risk) of* $\mathbf{w}$ *under* P *is defined as:*

$$R(\mathbf{w}) := \mathbb{E}_{\mathbf{x},y}\left[(y - \mathbf{w}^T\mathbf{x})^2\right] \tag{2.4}$$

$$= \int P(\mathbf{x}, y)(y - \mathbf{w}^T\mathbf{x})^2 d\mathbf{x}dy \tag{2.5}$$

The problem is that P is not known, and thus we can't directly optimize for R. Instead, we estimate R.

Definition 2.3.2 (Estimated Expected Error) *Let* D *be some labeled data, then estimated true risk (or empirical risk) is defined as:*

$$\hat{R}_D(\mathbf{w}) := \frac{1}{|D|}\sum_{(\mathbf{x},y)\in D}(y - \mathbf{w}^T\mathbf{x})^2 \tag{2.6}$$

Then, by the Law of large numbers, we know that $\hat{R}_D(\mathbf{w}) \xrightarrow[|D|\to\infty]{} R(\mathbf{w})$ for any fixed $\mathbf{w}$, *i.e.* the more data we have the better our approximation for R will be since it will approach the true value.

Finally we can optimize our empirical risk using our training data:

$$\hat{\mathbf{w}}_D := \arg\min_{\mathbf{w}} \hat{R}_D(\mathbf{w}) \tag{2.7}$$

ideally:

$$\mathbf{w}^* := \arg\min_{\mathbf{w}} R(\mathbf{w}) \tag{2.8}$$

However, it's not always the case that as we have more training data in D the optimal risk $\mathbf{w}^*$ approaches the empirical risk $\hat{\mathbf{w}}_D$ (this is not implied by the Law of large numbers alone), for this we need the stronger notion of uniform convergence.

Definition 2.3.3 (Uniform Convergence) *We say that* R *converges uniformely if*

$$\sup_{\mathbf{w}} \left|R(\mathbf{w}) - \hat{R}_D(\mathbf{w})\right| \xrightarrow[|D|\to\infty]{} 0 \tag{2.9}$$

Learning from Data The previous notions use the fact that $|D|$ must approach infinity, however we always deal with a finite amount of training samples and hence the following problem occurs:

Lemma 2.3.1 (Optimistic Estimate) *Given a data set D we have that:*

$$\mathbb{E}_D\left[\hat{R}_D(\hat{w}_D)\right] \leq \mathbb{E}_D\left[R_D(\hat{w}_D)\right] \tag{2.10}$$

Proof.

$$\begin{aligned}
\mathbb{E}_D\left[\hat{R}_D(\hat{\mathbf{w}}_D)\right] &= \mathbb{E}_D\left[\min_{\mathbf{w}} \hat{R}_D(\mathbf{w})\right] \\
&\leq \min_{\mathbf{w}} \mathbb{E}_D\left[\hat{R}_D(\mathbf{w})\right] && \textit{Jensen's Inequality} \\
&= \min_{\mathbf{w}} \mathbb{E}_D\left[\frac{1}{|D|}\sum_{i=1}^{|D|}(y_i - \mathbf{w}^T\mathbf{x_i})^2\right] \\
&= \min_{\mathbf{w}} \frac{1}{|D|}\sum_{i=1}^{|D|}\mathbb{E}_{(\mathbf{x_i},y_i)\sim P}\left[(y_i - \mathbf{w}^T\mathbf{x_i})^2\right] \\
&= \min_{\mathbf{w}} R(\mathbf{w}) \\
&\leq \mathbb{E}_D\left[R_D(\hat{w}_D)\right]
\end{aligned}$$

□

Lemma 2.3.1 tells us that the expected value of the expected estimated error is always less than the expected value of the true error. This is a problem, because we will always estimate a smaller error than what we actually have by using a finite training set. In order to avoid underestimating the prediction error, we will use two different data sets D_{train} and D_{test} from the same distribution $D_{train}, D_{test} \sim P$, then:

Lemma 2.3.2 (Correct Estimate) *Given $D := D_{train}$ and $V := D_{test}$, then:*

$$\mathbb{E}_{D,V}\left[\hat{R}_V(\hat{\mathbf{w}}_D)\right] = \mathbb{E}_D\left[R(\hat{\mathbf{w}}_D)\right] \tag{2.11}$$

Proof.

$$\begin{aligned}
\mathbb{E}_{D,V}\left[\hat{R}_V(\hat{\mathbf{w}}_D)\right] &= \mathbb{E}_D\left[\mathbb{E}_V\left[\hat{R}_V(\hat{\mathbf{w}}_D)\right]\right] && D, V \textit{ are i.i.d.} \\
&= \mathbb{E}_D\left[\mathbb{E}_V\left[\frac{1}{|V|}\sum_{i=1}^{|V|}(y_i - \hat{\mathbf{w}}_D^T\mathbf{x_i})^2\right]\right] \\
&= \mathbb{E}_D\left[\frac{1}{|V|}\sum_{i=1}^{|V|}\mathbb{E}_{(\mathbf{x_i},y_i)\sim V}\left[(y_i - \hat{\mathbf{w}}_D^T\mathbf{x_i})^2\right]\right] \\
&= \mathbb{E}_D\left[R(\hat{\mathbf{w}}_D)\right]
\end{aligned}$$

□

Lemma 2.3.2 tells us that if we use independent train and test (validation) sets, the expected value of the estimated error is the same as the expected value of the true error, and thus we will be able to estimate the correct error by using our test set D_{test} without having a wrong underestimation. This works because the two sets are independent, and thus we have an unbiased error. We must be careful to choose the test data in a way that it's actually independent from the training data. [1]

1: Test data samples might not be independent if drawn from:

- **Time series** data might contain time-correlated values, *e.g.* stocks, video, audio,...
- **Spatial** data might be correlated *e.g.* images.
- **Noise** might contain correlated data.

2.4 Cross Validation

We have analyzed the expected prediction error using D_{train} and D_{test} as samples from a distribution P. In practice, we are given a labeled dataset D of finite dimension, hence we can't sample the data from such a distribution. Recall our initial goal of finding a good model that optimizes goodness of fit given different parameters, *e.g.* different degrees for the polynomials used in the polynomial regression. Thus we have to find a way to pick D_{train} and D_{test} from D and a way to exploit them in order to evaluate the performance of a given model. This process is called *cross-validation* and there are different ways to apply it.

Monte Carlo Cross-Validation We split the dataset D into two disjoint sets $D = D_{train} \uplus D_{test}$ by picking some number of elements uniformly at random such that $D_{train} \subset D$ and the remaining elements will be in the test set $D_{test} = D \setminus D_{train}$. Then we train the model on the training set D_{train} and validate on the test set D_{test}. Lastly, we estimate the prediction error by averaging the test error over multiple random trials and we pick the best model.

Algorithm 2.2: Monte Carlo CV

foreach model $m = 1, ..., M$
  foreach repetition $r = 1, \ldots, R$
    $D = D_{train} \uplus D_{test}$ ▷ *Split training and test sets randomly*
    $\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \hat{R}_{D_{train}}(\mathbf{w})$ ▷ *Train model*
    $\hat{R}_m^{(r)} := \hat{R}_{D_{test}}(\hat{\mathbf{w}})$ ▷ *Save estimated error of repetition r and model m*
  end
end
return $\hat{m} = \arg\min_m \frac{1}{R} \sum_{r=1}^{R} \hat{R}_m^{(r)}$ ▷ *Pick model with smallest average error*

K-fold Cross-Validation We split the dataset D into k disjoint sets such that $D = D_{train}^{(1)} \uplus \cdots \uplus D_{train}^{(k)}$ and $D_{test}^{(i)} := D \setminus D_{train}^{(i)}$. Then, for each model, we train on each training fold and validate on each test fold. Lastly, we estimate the prediction error by averaging over multiple folds and we pick the best model.

Algorithm 2.3: K-fold CV

foreach model $m = 1, ..., M$
  foreach fold $i = 1, \ldots, k$ ***do***
    $D = D_{train}^{(i)} \uplus D_{test}^{(i)}$ ▷ *Split training and test sets using fold i*
    $\hat{\mathbf{w}}^{(i)} = \arg\min_{\mathbf{w}} \hat{R}_{D_{train}}^{(i)}(\mathbf{w})$ ▷ *Train model with fold i*
    $\hat{R}_m^{(i)} := \hat{R}_{D_{test}}^{(i)}(\hat{\mathbf{w}}^{(i)})$ ▷ *Save estimated error of fold i and model m*
  end
end
return $\hat{m} = \arg\min_m \frac{1}{k} \sum_{i=1}^{k} \hat{R}_m^{(i)}$ ▷ *Pick model with smallest average error*

What are the tradeoffs of choosing different sizes of k?

If k is too small:

- Risk of overfitting to the test set.
- Risk of using too little data for training.
- Risk of underfitting to the training set.

If k is too large:

- Better performance, usually $k = n$ works really well and it's called leave-one-out cross-validation *LOOCV*.
- Higher computational complexity.
- Risk of underfitting to training set.

2.5 Model Selection

Cross-validation is a useful tool for *model selection*, *i.e.* choosing the model which performs best on our task and, hopefully, generalizes well to real-world data. In case we have to select the best polynomial degree for a regression task, we can simply iterate with increasing degrees and check which model has the least expected error. However polynomial regression is not the only possible case, in facts we might also want to perform regression by choosing a basis function which is not a polynomial (*e.g.* a combination of trigonometrical, logarithmical and exponential transformations). In those cases, iterating with increasing degree is not an option. Here one would have to create a list of candidate models (without any ordering relation between them) and iterating it.

2.6 Regularization

Standard linear regression seeks to optimize the data fit by minimizing the mean squared error $\min_{\mathbf{w}} \hat{R}(\mathbf{w})$. This approach might work well if our data doesn't contain any outliers, but it may be problematic if our data are (a little bit) noisy. Consider a dataset with 1000 points. A polynomial with degree 1000 will interpolate over those points and hence will have a zero error on them. However, if a few points are outliers, our high-degree polynomial will have an artificial shape to fit the noise. This will lead to a large error to some points which are in proximity of this artificial shape and hence this model would not generalize well. This polynomial will have some large weights in order to interpolate the points and hence we can say that large weights are indicators of overfitting. Our goal is to keep weights reasonably small in order to avoid such a phenomenon. Thus, instead of just optimizing just for the mean squared error, we add an additional penalty on the size of the weights. This practice is called *regularization*.

Definition 2.6.1 (Regularization) *Given* penalty function C *and a* regularization parameter λ *we can define a* regularization problem *as:*

$$\min_{\mathbf{w}} \hat{R}(\mathbf{w}) + \lambda C(\mathbf{w}) \tag{2.12}$$

where λ is used to weight how much C should penalize $\mathbf{w}$.

Regularization seeks to find a balance between the goodness of fit and the magnitude of the weights. There are countless possible regularizers, here we present *ridge regression*, a popular solution that is widely adopted in practice.

Ridge Regression This type of regularization uses the squared norm $C(\mathbf{w}) := \|\mathbf{w}\|^2$ as the penalty function which has nice mathematical properties when we want to solve for $\mathbf{w}$:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \mathbf{w}^T \mathbf{x_i}\right)^2 + \lambda \|\mathbf{w}\|_2^2 \tag{2.13}$$

The solution to this problem can be found both via gradient descent:

Gradient Evaluation

$$\nabla_{\mathbf{w}} \left(\hat{R}(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2\right) = \nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) + \lambda \nabla_{\mathbf{w}} \|\mathbf{w}\|_2^2 \tag{2.14}$$

$$= \nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) + \lambda \nabla_{\mathbf{w}} (\mathbf{w}^T \mathbf{w}) \tag{2.15}$$

$$= \nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) + 2\lambda \mathbf{w} \tag{2.16}$$

GD Update Rule

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \left(\nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) + 2\lambda \mathbf{w}\right) \tag{2.17}$$

$$= (1 - 2\lambda\eta_t)\mathbf{w}_t - \eta_t \nabla \hat{R}(\mathbf{w}) \tag{2.18}$$

and via analytical solution:

$$\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y} \tag{2.19}$$

The best λ is typically chosen with cross validation over a logarithmically spaced list of candidates [2] .

2: $\lambda \in \{\ldots, 10^{-2}, 10^{-1}, 10, 10^1, 10^2, \ldots\}$

2.7 Standardization

In the previous section, we have seen that large weights often correspond to noise and are indicators of overfitting. For this reason, we introduced a term to penalize large weights. However, the idea that having smaller weights leads to more accurate models might not always be true. Consider an example where we have three features and those are in completely different magnitudes (*e.g.* the first feature is in the order of 10^4, the second in the order of 10^3 and the third one in the order of 10^0). If we penalize large weights we might come to a situation where all weights are similar (*e.g.* all close to one). However, since the features have a completely different magnitude, the first feature would have a much larger impact than the other ones, and this is undesirable since we would lose the information brought by the other features. A solution to this problem is using *standardization* to scale our data such that they have zero mean and unit variance.

$$\hat{\mu}_j = \frac{1}{n} \sum_{i=1}^{n} x_{i,j} \tag{2.20}$$

$$\hat{\sigma}_j^2 = \frac{1}{n} \sum_{i=1}^{n} (x_{i,j} - \hat{\mu}_j)^2 \tag{2.21}$$

$$\tilde{x}_{i,j} := \frac{x_{i,j} - \hat{\mu}_j}{\hat{\sigma}_j} \tag{2.22}$$

Classification 3

3.1 Binary Classification 15
3.2 Perceptron Algorithm 16
3.3 Stochastic Gradient Descent . 17
3.4 Support Vector Machine . . . 17
3.5 Feature Selection 18
3.6 Class Imbalance 21
3.7 Multi-class Classification . . 24

In the previous chapter we discussed regression, *i.e.* the problem of predicting a function $f : X \to Y$, where Y is a continuous set such as $\mathbb{R}$. Now we introduce *classification*, where the set Y is discrete. The high-level idea is assigning each point in X to a specific category. For example, the space X could represent the space of pictures and we want to find the best way to assign each picture to either the category *cat* or the category *dog*, depending on which animal is represented in the picture. In order to design such algorithms, we will combine some concepts learned in the previous chapter (*e.g.* gradient descent, regularization, the general idea of minimizing a loss function, ...) with new, *ad hoc* tools.

3.1 Binary Classification

Goal

Given a set of feature vectors $\mathbf{x_1}, \ldots, \mathbf{x_n}$ where $\mathbf{x_i} \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$), and a set of labels $y_1, \ldots, y_n$ where $y_i \in \{-1, +1\}$ (which can be represented as a vector $\mathbf{y} \in \{-1, +1\}^n$).

Output the coefficient vector $\mathbf{w} \in \mathbb{R}^d$ such that

$$f(\mathbf{x_i}) := \text{sign}(\mathbf{w}^T \mathbf{x_i}) = y_i, \quad \forall i \in \{1, \ldots, n\} \tag{3.1}$$

The intuition is that $\mathbf{w}$ represents the normal to a hyperplane (in 2d the norm to the separating line) that points in the direction of the positive labels. If the point $\mathbf{x_i}$ is on the same side as where the vector $\mathbf{w}$ points to, we will have a positive dot product $\mathbf{w}^T \mathbf{x_i}$ and thus a sign of +1, otherwise a negative dot product and a sign of -1.

0/1 Loss To find the optimal $\hat{\mathbf{w}}$ we could consider the approach of counting the number of samples that we got wrong using a loss function called 0/1 *loss* and then minimizing that value.

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell_{0/1}(\mathbf{w}; \mathbf{x_i}, y_i) \tag{3.2}$$

Definition 3.1.1 (0/1 Loss) *Given the coefficient vector* $\mathbf{w}$ *the current feature vector* $\mathbf{x_i}$ *and label* y_i*, the* 0/1 loss *is defined as:*

$$\ell_{0/1}(\mathbf{w}; \mathbf{x_i}, y_i) := \begin{cases} 0 & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} \geq 0 \\ 1 & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} < 0 \end{cases} \tag{3.3}$$

The problem is that $\ell_{0/1}$ is neither differentiable nor convex, and thus we cannot use our standard optimization method such as gradient descent.

3.2 Perceptron Algorithm

Since $\ell_{0/1}$ is not suitable for our purposes, we have to introduce a *surrogate loss* which is both informative and compatible with gradient descent. The perceptron algorithm uses the following loss function ℓ_P, which is similar to $\ell_{0/1}$, convex and differentiable.

Definition 3.2.1 (Perceptron Loss) *Given the coefficient vector* $\mathbf{w}$ *the current feature vector* $\mathbf{x_i}$ *and label* y_i*, the* perceptron loss *is defined as:*

$$\ell_P(\mathbf{w}; \mathbf{x_i}, y_i) := \max(0, -y_i \cdot \mathbf{w}^T \mathbf{x_i}) \tag{3.4}$$

The following objective function can now be optimized with gradient descent.

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell_P(\mathbf{w}; \mathbf{x_i}, y_i) \tag{3.5}$$

Gradient Evaluation

$$\nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \max(0, -y_i \cdot \mathbf{w}^T \mathbf{x_i}) \tag{3.6}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 0 & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} \geq 0 \\ -y_i \mathbf{x_i} & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} < 0 \end{cases} \tag{3.7}$$

GD Update Rule

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 0 & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} \geq 0 \\ -y_i \mathbf{x_i} & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} < 0 \end{cases} \tag{3.8}$$

Theorem 3.2.1 (Perceptron Convergence) *If the provided training dataset is linearly separable, then the perceptron algorithm will always find coefficients* $\mathbf{w}$ *that linearly separate the data.*

Note that while we use the perceptron loss on the training data, we still use the $\ell_{0/1}$ loss on the test data to compute the number of errors and evaluate the performance of our model. A drawback of the algorithm we have presented so far is that, in order to do a single weights update, we have to iterate over the whole dataset. This might be very inefficient for large datasets. Now we present the variant of the perceptron algorithm which is most widely used in practice. This variant uses *stochastic gradient descent* (see next section) in order to efficiently optimize the objective function.

Algorithm 3.1: Perceptron

$\mathbf{w}_0 \leftarrow 0$
foreach $t = 1, 2, ..., T$
  sample $(\mathbf{x_i}, y_i) \in_{u.a.r.} D_{train}$ ▷ *Sample uniformly at random* with replacement
  if $y_i \mathbf{w}^T \mathbf{x_i} \geq 0$
    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t$
  else
    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta_t y_i \mathbf{x_i}$
end
return $\mathbf{w}_{T+1}$

3.3 Stochastic Gradient Descent

Standard gradient descent is highly inefficient: if we have a lot of training samples we have to loop through them all just to take a small step towards the optimum. Instead, *stochastic* gradient descent samples a single data point uniformly at random (with replacement) from the training set at each step, and therefore it's much more efficient.

Algorithm 3.2: SGD

$w_0 \in_{u.a.r.} \mathbb{R}^d$
foreach $t = 1, 2, ..., T$
  sample $(\mathbf{x}, y) \in_{u.a.r.} D_{train}$ ▷ *Sample uniformly at random* with replacement
  $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}_t} \ell(\mathbf{w}_t; \mathbf{x}, y)$
end
return $\mathbf{w_{T+1}}$

Mini-Batch SGD Using stochastic gradient descent with only a single sample might have a large variance in the gradient estimate, and hence leads to slow convergence. To solve this problem and reduce the variance, instead of picking a single training sample, we will pick a small subset of the training data called *mini-batch*. Mini-batches will both have the advantage of a fast and stable convergence.

Algorithm 3.3: Mini-Batch SGD

$w_0 \in_{u.a.r.} \mathbb{R}^d$
foreach $t = 1, 2, ..., T$
  sample $D_{batch} \subseteq_{u.a.r.} D_{train}$ ▷ *Mini-batch uniformly at random* with replacement
  $\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} - \eta_t \frac{1}{|D_{batch}|} \sum_{(\mathbf{x},y) \in D_{batch}} \nabla \ell(\mathbf{w_t}; \mathbf{x}, y)$
end
return $\mathbf{w_{T+1}}$

3.4 Support Vector Machine

The perceptron algorithm is an efficient method for binary classification and it always finds coefficients $\mathbf{w}$ that linearly separate the data if such coefficients exist. But what if there are *multiple* vector coefficients $\mathbf{w}$ that linearly separate the data? The perceptron algorithm will choose an arbitrary hyperplane. However, different hyperplanes can be more or less

error-prone than others. For example, if the line we pick is close to one of the two clusters of data, it will be more sensitive to noise than another one which keeps a larger margin between clusters. The *support vector machine* algorithm uses an objective function that maximizes the margin between the separating hyperplane and the data. With this method, lines close to the clusters of data are penalized and therefore noise resistance is increased. This is obtained by introducing a new loss function and applying regularization. [1]

1: Since we are using regularization remember that we have to standardize our data.

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell_H(\mathbf{w}; \mathbf{x_i}, y_i) + \lambda \|\mathbf{w}\|_2^2 \tag{3.9}$$

Definition 3.4.1 (Hinge Loss) *Given the coefficient vector* $\mathbf{w}$ *the current feature vector* $\mathbf{x_i}$ *and label* y_i*, the* hinge loss *is defined as:*

$$\ell_H(\mathbf{w}; \mathbf{x_i}, y_i) := \max\{0, 1 - y_i \mathbf{w}^T \mathbf{x_i}\} \tag{3.10}$$

Gradient Evaluation

$$\nabla_{\mathbf{w}} \hat{R}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \max\{0, 1 - y_i \mathbf{w}^T \mathbf{x_i}\} + \lambda \nabla_{\mathbf{w}} \|\mathbf{w}\|_2^2 \tag{3.11}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 0 & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} \geq 1 \\ -y_i \mathbf{x_i} & \text{if } y_i \cdot \mathbf{w}^T \mathbf{x_i} < 1 \end{cases} + 2\lambda \mathbf{w} \tag{3.12}$$

GD Update Rule [2]

2: Usually a good choice for the learning rate is $\eta_t = \frac{1}{\lambda t}$, where λ is found using cross-validation.

$$\mathbf{w}_{t+1} \leftarrow (1 - 2\eta_t \lambda)\mathbf{w}_t - \eta_t \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 0 & \text{if } y_i \cdot \mathbf{w_t}^T \mathbf{x_i} \geq 1 \\ -y_i \mathbf{x_i} & \text{if } y_i \cdot \mathbf{w_t}^T \mathbf{x_i} < 1 \end{cases} \tag{3.13}$$

SGD Update Rule

$$\mathbf{w}_{t+1} \leftarrow (1 - 2\eta_t \lambda)\mathbf{w}_t + \eta_t y_t \mathbf{x_t}, \quad [y_t \mathbf{w_t}^T \mathbf{x_t} < 1] \tag{3.14}$$

Similarly to the perceptron algorithm, we don't use the hinge loss for the validation of our model but we would use the target performance metric (*e.g.* the number of mistakes with the 0/1 loss).

3.5 Feature Selection

The models we have presented so far are trained with some feature vectors $\mathbf{x_1}, \ldots, \mathbf{x_n}$, where $x \in \mathbb{R}^d$. If the dimension d of a feature vector is high (*i.e.* there are many parameters) our model might take a long time to train. In many cases, some features in a feature vector are redundant and don't bring any useful information: keeping those features is not desirable since they make our model less efficient without improving its performance. For this reason, it's crucial to find a way to select only the important features. The optimization process of selecting the best features is called *feature selection* and, in general, it's a very difficult combinatorial problem. In this section, we will explore some heuristics to approach

the problem. Before we present them, it's important to introduce the preliminary notion of *feature error*.

Feature Error

Definition 3.5.1 (Feature Selection) *Given a set of features $V = \{1, \dots, d\}$ with $S \subseteq V$ of cardinality k, and one feature vector $\mathbf{x_i} = [x_{i,1}, \dots, x_{i,d}]$, then the* feature selection *of that feature vector is defined as*

$$\mathbf{x_i}^{(S)} := [x_{i,1}, \dots, x_{i,k}] \tag{3.15}$$

When using this feature selection, the associated coefficient vector is given by:

$$\hat{\mathbf{w}}^{(S)} := \arg\min_{\mathbf{w}^{(S)}} \frac{1}{n} \sum_{i=1}^{n} \ell(\mathbf{w}^{(S)}; \mathbf{x_i}^{(S)}, y_i) + \lambda \|\mathbf{w}\|_2^2 \tag{3.16}$$

A feature selection just picks a sparse version of the initial feature vector that is then reduced to a lower dimensional vector, thus both $\mathbf{w}^{(S)}$ and $\mathbf{x_i}^{(S)}$ will be a lower-dimensional version of $\mathbf{w}$ and $\mathbf{x_i}$ respectively. We will now be able to define the *feature error* as:

Definition 3.5.2 (Feature Error) *Given $S \subseteq V$, and a coefficient vector $\hat{\mathbf{w}}^{(S)}$, the* feature error *is the cross validation prediction accuracy of $\hat{\mathbf{w}}^{(S)}$ denoted as $\hat{L}(S)$.*

The feature error allows us to measure the error of a subset of features. We can use this concept to pick different sets S take the one that gives the least feature error $\hat{L}(S)$. The first, naive idea, would be to test all possible subsets of S and pick the best one. However, this method is highly inefficient and thus we will consider more clever methods.

Greedy Forward Selection The idea of *greedy forward selection* is starting with an empty set S and always picking the remaining feature that decreases the error the most until the error increases.

Algorithm 3.4: Greedy Forward Selection

```
1 S ← ∅
2 E_0 ← ∞
3 foreach i = 1, ..., d
4     s_i := arg min_{j∈V\S} L̂(S ∪ {j})        ▷ Find best element to add
5     E_i ← L̂(S ∪ {s_i})                       ▷ Compute error
6     if E_i > E_{i−1} break                   ▷ Stop if new best element increases error
7     else S ← S ∪ {s_i}                       ▷ Otherwise add new best element and continue
8 end
9 return S
```

The advantage of this algorithm is that it's relatively fast if we have only a few features that are important and many that are not. However, it cannot handle dependent features well since it might get stuck in a sub-optimal solution, especially if almost all features are necessary.

Greedy Backward Selection The idea of *greedy backward selection* is starting with the entire set of features S, and always removing the remaining feature that decreases the error the most.

Algorithm 3.5: Greedy Backward Selection

```
1 S ← {1, ..., k}
2 E_{d+1} ← ∞
3 foreach i = d, ..., 1
4     s_i := arg min_{j∈S} L̂(S \ {j})        ▷ Find best element to remove
5     E_i ← L̂(S \ {s_i})                     ▷ Compute error
6     if E_i > E_{i+1} break                 ▷ Stop if removing element increases error
7     else S ← S \ {s_i}                     ▷ Otherwise remove new best element and continue
8 end
9 return S
```

This selection can handle dependent features much better. If almost all features are important and only a few can be removed this algorithm might work better than forward selection.

Joint Selection Both greedy feature selection methods are expensive if we have a large dataset, furthermore neither method guarantees that an optimal solution is reached. It would be ideal if we could find all the important features by solving a single optimization problem. The idea is that, as we have seen before, the coefficient vector $\mathbf{w}^{(S)}$ will be a lower-dimensional version of a sparse $\mathbf{w}$, thus if we limit the sparsity of $\mathbf{w}$ we would reduce the highest number of features. This method is called *joint selection*.

$$\hat{\mathbf{w}} := \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x_i})^2 \; s.t. \; \|\mathbf{w}\|_0 \leq k \tag{3.17}$$

Definition 3.5.3 (L_0-Norm) *Let* $\mathbf{w}$ *be a vector, then the* L_0-Norm *is:*

$$\|\mathbf{w}\|_0 := \textit{number of non-zeros in } \mathbf{w} \tag{3.18}$$

Thus by constraining $\|\mathbf{w}\|_0$ to be smaller than k, we will constrain the number of features to be at most k. However, optimizing $\|\mathbf{w}\|_0$ is a hard combinatorial problem and it cannot be solved easily.

Lasso Regression Previously, we have seen how the perceptron algorithm uses a surrogate loss ℓ_P such that $\ell_{0/1}$ can be approximated and optimized. In this case, we will use a similar method, but instead of working with a surrogate loss we will use a surrogate convex regularization term. The idea is that instead of limiting the sparsity of the vector $\mathbf{w}$ to be at most k, we will penalize large weights of $\mathbf{w}$ in the following way:

$$\hat{\mathbf{w}} := \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x_i})^2 + \lambda \|\mathbf{w}\|_1 \tag{3.19}$$

Definition 3.5.4 (L_1-Norm) *Let* $\mathbf{w}$ *be a vector, then the* L_1-Norm [a] *is:*

$$\|\mathbf{w}\|_1 := \sum_{i=1}^{d} |w_i| \tag{3.20}$$

[a] This norm is convex and thus easy to optimize for.

Where the L_1-Norm will penalize large weights and thus maximize the sparsity[3] of $\mathbf{w}$. This regression method is called *Lasso Regression*.

3: This idea of using the L_1-Norm to maximize sparsity of the coefficient vector is very important and used thoroughly in machine learning.

One clear advantage of this method is that it's faster. We will train the model and at the same time select the best feature by maximizing sparsity. However, this method only works for linear models, where the greedy methods are slower but apply to any model.

3.6 Class Imbalance

Sometimes it's possible that we are given an unbalanced dataset, which means that the labels $y_1, \dots, y_n \in \{-1, +1\}$ of the feature vectors are not properly divided in two sets of equal size representing $+1$ and -1. More formally, given $P := \{y_i \mid y_i = +1\}$ and $M := \{y_i \mid y_i = -1\}$ we have either $|P| \gg |M|$ or $|P| \ll |M|$. We will refer to the set with more elements as the *majority class* and to the set with fewer elements to the *minority class*.

Example 3.6.1 Some examples of applications suffer from class imbalance:

- **Reccomender systems**, which suggest items of interest to the user (*e.g.* Advertisement, Movies, Books, ...). In this context the data about what the user likes is much less than the data of what the user doesn't like.
- **Fraud detection**, the number of fraud transactions is (usually) much less than the number of normal transactions.
- **Medical applications**, the number of people with a disease is much less than those without.

There are a few issues with class imbalance. If we use the fraction of correctly labeled elements (accuracy) as our metric to test the performance, even if our classifier doesn't work well, we will label most of the elements correctly since the number of labels in the minority class will contribute almost nothing to the error. Also, during training, the minority class may be ignored for optimization since it will contribute little to the empirical risk. Thus we will have to find a better way both to train and to test our classifier.

Naive Classifier One easy solution could be to subsample the data of the majority class (*e.g.* by removing them uniformly at random) until it contains the same number of samples of the minority class. The advantage of this approach is that we get a smaller dataset. However, we throw away a lot of useful information. The opposite solution is to *upsample* the minority class by duplicating data (possibly with some random perturbation) until we obtain the same number of data as in the majority

class. The advantage is that we make use of all data but the issue is that adding perturbation might give us inaccurate data and the dataset will be much larger and thus slower to train. Those naive solutions are not optimal but deal with both the training and testing problem.

Cost-Sensitive Classifier Another solution could be giving more importance to the minority class during training. In order to do this, we will define a slightly modified loss function that can be applied to any of the cost functions that we have seen.

Definition 3.6.1 (Cost-Sensitive Loss) *Given a loss function $\ell_\star(\mathbf{w}; \mathbf{x}, y)$ and a scalar $c_y \in \mathbb{R}_{>0}$ that depends on the label y we will define it's cost-sensitive loss as:*

$$\ell_\star^{(CS)}(\mathbf{w}; \mathbf{x}, y) := c_y \ell_\star(\mathbf{w}; \mathbf{x}, y) \tag{3.21}$$

Using the cost sensitive loss we can redefine the empirical risk as:

$$\hat{R}(\mathbf{w}; c_+, c_-) = \frac{1}{n} \sum_{i:y_i=+1} c_+ \ell_\star(\mathbf{w}; \mathbf{x}, y) + \frac{1}{n} \sum_{i:y_i=-1} c_- \ell_\star(\mathbf{w}; \mathbf{x}, y) \tag{3.22}$$

Where c_+ is the cost that we put on the data in the majority class and c_- the cost of the data in the minority class. Note that this empirical risk has the following property:

$$\forall \alpha > 0 : \ \alpha \hat{R}(\mathbf{w}; c_+, c_-) = \hat{R}(\mathbf{w}; \alpha c_+, \alpha c_-) \tag{3.23}$$

Thus if we let $\alpha := \frac{1}{c_-}$ we can redefine the empirical risk as:

$$\hat{R}(\mathbf{w}; \frac{c_+}{c_-}, 1) = \hat{R}(\mathbf{w}; c) \tag{3.24}$$

which removes the redundancy of using two different costs and thus we can only use $c := \frac{c_+}{c_-}$ as a weighting factor. Then if $c > 1$ we will give more importance to the class where $y_i = +1$ and if $c < 1$ we will give more importance to the class where $y_i = -1$.

Threshold Classifier Instead of using a cost-sensitive classifier can also use another option which is to use standard classifier and change classification threshold used to find the predicted value for some value of $\tau \in \mathbb{R}$.

$$\hat{y} = \text{sign}(\mathbf{w}^T \mathbf{x} + \tau) \tag{3.25}$$

This method will move the boundary of the classifier and if moved in the right direction it might label correctly more data in the minority class.

Testing Unbalanced Classifiers The cost-sensitive loss and the threshold classifier are different options that we can take to solve the problem of training an unbalanced classifier (*i.e.* to find $\mathbf{w}$), however we still have to find a way to properly test it and to choose the values of c for the cost-sensitive and τ for the threshold classifier.

Given a test set, *i.e.* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ with labels $y_1, \ldots, y_n$, and the trained classifier $\mathbf{w}$, let $\hat{y}_i = \text{sign}(\mathbf{w}^T \mathbf{x_i})$ (or $\hat{y}_i = \text{sign}(\mathbf{w}^T \mathbf{x_i} + \tau)$ if using the threshold classifier) be the predicted value for feature x_i. Then the usual way to check the accuracy is to count the fraction of correctly classified elements.

$$accuracy = \frac{1}{n} \sum_{i=1}^{n} 1_{y_i = \hat{y}_i} \tag{3.26}$$

Where the accuracy should be as close as possible to 1. However, as we have discussed before, this method doesn't work well because we give the same importance to errors in the minority and the majority classes. To change our definition of accuracy we have to define the notion of *true/false positive/negative*.

	Positive Label	Negative Label
Positive Prediction	$TP := \sum_{i=1}^{n} 1_{\hat{y}_i = +1 \wedge y_i = +1}$	$FP := \sum_{i=1}^{n} 1_{\hat{y}_i = +1 \wedge y_i = -1}$
Negative Prediction	$FN := \sum_{i=1}^{n} 1_{\hat{y}_i = -1 \wedge y_i = +1}$	$TN := \sum_{i=1}^{n} 1_{\hat{y}_i = -1 \wedge y_i = -1}$

Where the number of positive labels is $n_+ := TP + FN$, the number of negative labels is $n_- := FP + TN$, the number of positive prediction is $p_+ := TP + FP$ and the number of negative prediction is $p_- := FN + TN$ where $n_+ + n_- = p_+ + p_- = n$. Furthermore the number of correctly classified elements is $t := TP + TN$ and the number of wrongly classified elements is $f := FN + FP$. Then using the previous definition we can build different types of metric. Note that we use the convention that the minority class will use positive labels.

- **Accuracy**:

$$accuracy = \frac{t}{n} = \frac{TP + TN}{TP + TN + FP + FN} \in [0, 1] \tag{3.27}$$

  which is the same accuracy (*i.e.* number of correctly classified elements) that we have defined before.
- **Precision (or TPR)**

$$precision = \frac{TP}{p_+} = \frac{TP}{TP + FP} \in [0, 1] \tag{3.28}$$

  which measures the fraction of elements that have been correctly predicted positive out of the ones that are predicted positive.
- **Recall**

$$recall = \frac{TP}{n_+} = \frac{TP}{TP + FN} \in [0, 1] \tag{3.29}$$

  which measures the fraction of correctly predicted positive out of the one that are labeled positive.
- **FPR**

$$FPR = \frac{FP}{n_-} = \frac{FP}{TN + FP} \in [0, 1] \tag{3.30}$$

  which measures the false positive rate.

- **F1 Score**

$$F1 = \frac{2TP}{2TP + FP + FN} = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} \in [0, 1] \tag{3.31}$$

  which considers both precision and recall.

Then, as always, we can use cross-validation to pick different models $\mathbf{w}$ and values for c if using the cost-sensitive classifier or τ if using the threshold classifier. However, instead of using accuracy, we can use the newly defined metrics like the F1 score to test the performance of the unbalanced model.

There are many ways to use the different metrics, one of them is to plot for different algorithms on the x,y-axis $(recall, precision)$ in what is called the *precision recall curve*, then the goal is to get as close as possible to $(1, 1)$. Another way is to plot on the x,y-axis (FPR, TPR) in what is called the *ROC Curve*, where the goal is to get as close as possible to $(0, 1)$.

Theorem 3.6.1 (PR vs ROC) *A model $\mathbf{w}_1$ dominates[a] a model $\mathbf{w}_2$ in terms of the ROC Curve iff $\mathbf{w}_1$ dominates model $\mathbf{w}_2$ in terms of the PR curve.*

[a] Dominate means a curve that is always above the other.

Note that if the area under either the ROC or PR curve (AUC) is less than $\frac{1}{2}$ then there is something wrong, probably the labels are swapped.

3.7 Multi-class Classification

In the previous sections, we considered binary classification, *i.e.* the presence of two classes *i.e.* $y_i \in \{-1, +1\}$. However, in some cases, we might want to consider more than two classes. In such scenarios, the algorithms we studied for the binary classification task are no longer applicable, at least directly. This problem is called *multi-class* classification.

Goal

Given a set of feature vectors $\mathbf{x_1}, \ldots, \mathbf{x_n}$ where $\mathbf{x_i} \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$), and a set labels $y_1, \ldots, y_n$ where $y_i \in C = \{1, 2, \ldots, |C|\}$ (which can be represented as a vector $\mathbf{y} \in C^n$).

Output the coefficient vector $\mathbf{w} \in \mathbb{R}^d$ such that

$$f(\mathbf{x_i}) := \text{sign}(\mathbf{w}^T \mathbf{x_i}) = y_i, \quad \forall i \in \{1, \ldots, n\} \tag{3.32}$$

One-Vs-All One easy way to do multi-class classification without building a new classifier from scratch is to transform the labels of the data such that we can reuse the well known binary classifier. To do so we will pick each class individually and compare it to all others. This method is called *one-vs-all* multi-class classification. More formally let $c \in C$ be the

current class, then we will redefine:

$$\forall i: \ \tilde{y}_i^{(c)} = \begin{cases} +1 & \text{if } y_i = c \\ -1 & \text{otherwise} \end{cases} \tag{3.33}$$

and train $|C|$ different binary classifiers, where each one of them is responsible to detect a single class in C. The problem is that it's possible that a feature vector $\mathbf{x}$ is detected as part of more than one class if we use the standard prediction method $\hat{y}^{(c)} = \text{sign}(\mathbf{w}^T\mathbf{x})$ (for class $c \in C$). To solve this problem we have to understand the notion of *confidence*.

Definition 3.7.1 (Confidence) *Given a trained model $\mathbf{w}^{(c)}$ for a class $c \in C$ and a feature vector $\mathbf{x}$, then the* confidence *of this class is defined as:*

$$f^{(c)}(\mathbf{x}) := (\mathbf{w}^{(c)})^T\mathbf{x} \tag{3.34}$$

Geometrically the confidence is higher if the feature vector x is further away from the decision boundary defined by $\mathbf{w}$. Then instead of using the sign function to check whether the feature vector is on one side of the decision boundary we use the confidence as a number that tells us how much away from the decision boundary is $\mathbf{x}$, which gives us a metric to compare the classifier from different classes. Thus our new prediction method will be defined by the classifier with the highest confidence.

$$\hat{y} = \arg\max_{c \in C} f^{(c)}(x) = (\mathbf{w}^{(c)})^T\mathbf{x} \tag{3.35}$$

We have to be careful when computing different $\mathbf{w}$, the sign function is invariable to scale *i.e.* $\forall \alpha > 0: \ \text{sign}(\alpha\mathbf{w}^T\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x})$, however the same doesn't hold for the confidence. To solve this problem we have to either normalize the weight vector $\mathbf{w} \leftarrow \frac{\mathbf{w}}{\|\mathbf{w}\|_2}$, or use regularization to force all the weight vector to be small and have approximately the same magnitude $\|\mathbf{w}\|_2$. Another problem is that by isolating a single class and comparing it to all others we will have an unbalanced classification problem, and thus we will have to use cost-sensitive or threshold classifiers. Lastly, if one class is not linearly separable from all others this method will fail.

One-Vs-One If some classes are not linearly separable, one-vs-all will fail to separate them appropriately. Instead it makes more sense to compare only two classes at the time and possibly use a non-linear classifier between them. This method is called *one-vs-one* multi-class classification. Let $c_1, c_2 \in C$ be the classes in comparison, then we will redefine:

$$\forall i: \ \tilde{y}_i^{(c_1,c_2)} = \begin{cases} +1 & \text{if } y_i = c_1 \\ -1 & \text{if } y_i = c_2 \\ & \textit{otherwise ignore sample } i \textit{ if } y_i \notin \{c_1, c_2\} \end{cases} \tag{3.36}$$

and then train $\frac{|C|(|C|-1)}{2}$ binary classifiers (one for each pair). Here we don't need the notion of confidence, but instead the class with the highest number of positive prediction wins. The methods has the disadvantage

that it needs to train more classifiers, however it doesn't suffer from class imbalance and can handle non-linearly separable data.

Multi-class SVM In both previous methods, we had to train more than one classifier, however, by taking inspiration from the confidence in the one-vs-all method, we can modify the loss function of the SVM algorithm to handle multi-class classification.

Definition 3.7.2 (Multi-class Hinge Loss) *Given a feature vector* $\mathbf{x}$*, its label* y *and weight vectors* $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(|C|)}$ *we can define the* multi-class hinge loss *function is defined as:*

$$\ell_{MC-H}\left(\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(|C|)}; \mathbf{x}, y\right) = \max\left\{0, 1 + \max_{c \in C \setminus \{y\}} (\mathbf{w}^{(c)})^T \mathbf{x} - (\mathbf{w}^{(y)})^T \mathbf{x}\right\} \tag{3.37}$$

The key idea is that as in one-vs-all we keep $|C|$ weight vectors, then if we evaluate the confidence on the correct class y it must be higher than the confidence on all other classes by at least a margin (*e.g.* 1), *i.e.*:

$$\forall c \in C \setminus \{y\} : (\mathbf{w}^{(c)})^T \mathbf{x} > (\mathbf{w}^{(y)})^T \mathbf{x} + 1 \tag{3.38}$$

$$\iff \forall c \in C \setminus \{y\} : (\mathbf{w}^{(c)})^T \mathbf{x} - (\mathbf{w}^{(y)})^T \mathbf{x} > 1 \tag{3.39}$$

$$\iff \max_{c \in C \setminus \{y\}} (\mathbf{w}^{(c)})^T \mathbf{x} - (\mathbf{w}^{(y)})^T \mathbf{x} > 1 \tag{3.40}$$

Thus if the condition of Equation 3.40 $\star$ is satisfied the gradient of the loss function simplifies by a lot.

$$\nabla_{\mathbf{w}^{(c)}} \ell_{MC-H}\left(\mathbf{w}^{(1:|C|)}; \mathbf{x}, y\right) = \begin{cases} 0 & \textit{if } (\star) \ \vee (c \neq y \wedge c \neq \arg\max_{j \in C})(\mathbf{w}^{(j)})^T \mathbf{x} \\ -x & \text{if } \neg(\star) \wedge c = y \\ +x & \textit{otherwise} \end{cases} \tag{3.41}$$

Confusion Matrices We have seen that for unbalanced datasets we can use true/false positive/negative rates to select a better testing metric. When we are dealing with multi-class datasets we can create a square matrix that compares each predicted label with each true label and counts for each pair the number of predictions. Then the correct number of predicted labels will be counted on the diagonal of the matrix and everything that is not on the diagonal will be wrongly predicted labels.

Kernels 4

4.1 Feature Explosion Problem . 27
4.2 Polynomial Kernels 29
4.3 Kernelized Perceptron 30
4.4 Kernel Properties 31
4.5 Infinite Dimensional Kernels 32
4.6 Kernelized SVM 34
4.7 Kernelized Linear Regression 34

4.1 Feature Explosion Problem

Both in linear regression and in classification we can fit non linear (*e.g.* polynomial) functions to our data by mapping feature vectors $\mathbf{x_1}, \dots, \mathbf{x_n}$ to some non linear function $\phi(\mathbf{x_i}) = \tilde{\mathbf{x}}_\mathbf{i}$. This transformation returns a new set of data $\tilde{\mathbf{x}}_\mathbf{1}, \dots, \tilde{\mathbf{x}}_\mathbf{n}$ which can be optimized with a standard squared loss. Notice that $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ and since most of the times we wish to use more complicated features than the initial ones, we often have $d' > d$.

Example 4.1.1 In a real world scenario we might have $d = 10000$, *i.e.* 10000 features or 10000 dimensional vectors, which cannot be fitted with a linear function. We decide to use a polynomial of degree $m = 2$, which is the smallest possible polynomial (besides linear) that we can use. Thus, we have to evaluate our new feature vectors $\phi(\mathbf{x_i}) = \tilde{\mathbf{x}}_\mathbf{i} = [x_{i_1}^2, \dots, x_{i_d}^2, x_{i_1} \cdot x_{i_2}, \dots, x_{i_{d-1}} \cdot x_{i_d}]^T$, notice that $\tilde{\mathbf{x}}_\mathbf{i} \in \mathbb{R}^{d'}$ where $d' = 100000000$.

From the previous example we observe that in facts, even with a small degree polynomial, if we have a lot of features, we might have feature explosion from d to d^k features, and thus often $d' \gg d$ which is very computationally inefficient. The use of *kernel functions* will help us to solve this problem. In their essence, kernels allow us to exploit the benefits brought by a larger amount of features without paying for their overhead. In order to understand the core concepts of kernel methods, we introduce the following lemma.

Lemma 4.1.1 (Linear Optimum) *Given some labels y_i and some feature vectors $\mathbf{x_i}$, we can always find some scalars $\alpha_i \in \mathbb{R}$ for $i \in \{1, \dots, n\}$ such that we can represent the optimum $\hat{\mathbf{w}}$ as a linear combination:*

$$\hat{\mathbf{w}} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x_i} \tag{4.1}$$

Proof (Handwavy). We will give a handwavy proof for the specific cases of the perceptron and SVM algorithms. Recall that we can obtain the optimum $\hat{\mathbf{w}}$ with stochastic gradient descent in the following way:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \eta_t y_t \mathbf{x_t}, \quad [y_t \mathbf{w_t}^T \mathbf{x_t} < 0] \qquad \textit{Perceptron} \tag{4.2}$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t (1 - 2\lambda\eta_t) + \eta_t y_t \mathbf{x_t}, \quad [y_t \mathbf{w_t}^T \mathbf{x_t} < 1] \qquad \textit{SVM} \tag{4.3}$$

Consider the specific case of the SGD for the perceptron, after some time

T we will have

$$\begin{aligned}
\hat{\mathbf{w}} &= \mathbf{w}_{T+1} && (4.4)\\
&= \mathbf{w}_T + \eta_T y_T \mathbf{x_T} && \textit{SGD} \quad (4.5)\\
&= \mathbf{w}_{T-1} + (\eta_{T-1} y_{T-1} \mathbf{x_{T-1}}) + (\eta_T y_T \mathbf{x_T}) && \textit{SGD unroll twice} \quad (4.6)\\
&\vdots\\
&= \underset{=0}{\mathbf{w}_0} + (\eta_1 y_1 \mathbf{x_1}) + \cdots + (\eta_T y_T \mathbf{x_T}) && \textit{SGD unroll } T \textit{ times} \quad (4.7)\\
&= \sum_{i=1}^{n} \alpha_i y_i \mathbf{x_i} && \textit{Group same } y_i \mathbf{x_i} \quad (4.8)
\end{aligned}$$

Where α_i will be the sum of the learning rates η from the same terms $y_i \mathbf{x_i}$. The proof of the linear optimum for SVM is analogous. □

We will now see how Lemma 4.1.1 will help us to solve the problem of feature explosion. The basic idea is that instead of optimizing for the best $\hat{\mathbf{w}} \in \mathbb{R}^{d'}$ we want to find a way to optimize for $\hat{\boldsymbol{\alpha}} \in \mathbb{R}^n$. If we have feature explosion clearly $n \ll d'$ and thus the problem will be much less computationally expensive.

Perceptron Reformulation By using Lemma 4.1.1 we can redefine the optimum of the perceptron method by parametrizing it by $\boldsymbol{\alpha}$ and obtaining a dual optimization problem.

$$\begin{aligned}
&\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, -y_i \mathbf{w}^T \tilde{\mathbf{x}}_\mathbf{i}\} && (4.9)\\
&= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, -y_i \left(\sum_{j=1}^{n} \alpha_j y_j \tilde{\mathbf{x}}_\mathbf{j}\right)^T \tilde{\mathbf{x}}_\mathbf{i}\right\} && \textit{Lemma 4.1.1} \quad (4.10)\\
&= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, -\sum_{j=1}^{n} \alpha_j y_i y_j (\tilde{\mathbf{x}}_\mathbf{i}{}^T \tilde{\mathbf{x}}_\mathbf{j})\right\} && (4.11)\\
&= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, -\sum_{j=1}^{n} \alpha_j y_i y_j (\phi(\mathbf{x_i})^T \phi(\mathbf{x_j}))\right\} && (4.12)
\end{aligned}$$

Definition 4.1.1 (Kernel Function) *Let $\mathbf{x_i}, \mathbf{x}_j \in \mathbb{R}^d$ be two vectors and $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ be map, then the* kernel function $\mathrm{k} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ *is defined as:*

$$\mathrm{k}(\mathbf{x_i}, \mathbf{x_j}) := \phi(\mathbf{x_i})^T \phi(\mathbf{x_j}) \quad (4.13)$$

Then using the previous reformulation and the notion of kernel function we can rewrite the dual optimization problem as:

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, -\sum_{j=1}^{n} \alpha_j y_i y_j \, \mathrm{k}(\mathbf{x}, \mathbf{x}')\right\} \quad (4.14)$$

Now, even if we have reduced the problem to finding $\hat{\boldsymbol{\alpha}} \in \mathbb{R}^n$ instead of $\hat{\mathbf{w}} \in \mathbb{R}^{d'}$, there is still the problem that computing the dot product $\phi(\mathbf{x})^T \phi(\mathbf{x}')$ of the kernel function is very expensive (since $\phi(\mathbf{x})$ is also of dimension d'). The most important part of this section is to realize

that by using some clever tricks we can compute the kernel function $k(\mathbf{x_i}, \mathbf{x_j})$ without computing the dot product of dimension d' and neither the function $\phi(\mathbf{x})$.

4.2 Polynomial Kernels

Homogeneous Polynomial Kernel

Lemma 4.2.1 (Homogeneous Polynomial Kernel) *Let $\mathbf{x_i}, \mathbf{x}_j \in \mathbb{R}^d$. The* homogeneous polynomial kernel *of degree m, that corresponds to the feature space spanned by all products of* exactly[a] *m attributes, can be easily evaluated and is:*

$$k_m(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i}^T \mathbf{x_j})^m \tag{4.15}$$

[a] The dimensionality of this feature space is $d' := d^m$.

Lemma 4.2.1 tells us that we never have to actually evaluate the dot product $\phi(\mathbf{x_i})^T \phi(\mathbf{x_j})$, and thus the computation can be done much more efficiently.

Example 4.2.1 (Homogeneous Quadratic Kernel ($m = d = 2$))
Let $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ be defined as

$$\phi(\mathbf{x}) = \phi([x_1, x_2]) := [x_1^2, x_2^2, \sqrt{2}x_1x_2]^T = \tilde{\mathbf{x}} \tag{4.16}$$

which maps a $d' = 2$ dimensional feature vector to a $d' = 3$ dimensional feature vector. Then, given 2 feature vectors $\mathbf{x_i}, \mathbf{x_j}$ we can compute the kernel function in 2 ways:

$$\begin{aligned} k_2(\mathbf{x_i}, \mathbf{x_j}) &:= \phi(\mathbf{x_i})^T \phi(\mathbf{x_j}) && (4.17)\\ &= x_{i,1}^2 x_{j,1}^2 + x_{i,2}^2 x_{j,2}^2 + 2x_{i,1}x_{i,2}x_{j,1}x_{j,2} && (4.18)\\ &= (x_{i,1}x_{j,1} + x_{i,2}x_{j,2})^2 && (4.19)\\ &= (\mathbf{x_i}^T \mathbf{x_j})^2 && (4.20) \end{aligned}$$

In Equation 4.18 if we compute the function and the dot product in a standard way we have to do 2 additions and 3+3+4=10 multiplications, where in Equation 4.19, which is the one that can be directly derived with Lemma 4.2.1, only 1 addition and 3 multiplications.

Inhomogeneous Polynomial Kernel

Lemma 4.2.2 (Inhomogeneous Polynomial Kernel) *Let $\mathbf{x_i}, \mathbf{x}_j \in \mathbb{R}^d$. The* inhomogeneous polynomial kernel *of degree m, that corresponds to the feature space spanned by all products of* at most[a] *m attributes, can be easily evaluated and is:*

$$k_m(\mathbf{x_i}, \mathbf{x_j}) = (c + \mathbf{x_i}^T \mathbf{x_j})^m \tag{4.21}$$

[a] The dimensionality of this feature space is $d' := \binom{d+m}{m} = \mathcal{O}(d^m)$.

Proof. Let $\mathbf{n} = [n_0, \dots, n_d]^T$ where $|\mathbf{n}| = n_1 + \cdots + n_d$, then:

$$\begin{aligned}
\mathrm{k}_m(\mathbf{x}, \mathbf{y}) &= (c + \mathbf{x}^T\mathbf{y})^m && (4.22)\\
&= (c + x_1y_1 + \cdots + x_dy_d)^m && (4.23)\\
&= \sum_{|\mathbf{n}|=m} \binom{m}{\mathbf{n}} c^{n_0}(x_1y_1)^{n_1}(x_2y_2)^{n_2}\cdots(x_dy_d)^{n_d} && (4.24)\\
&= \sum_{|\mathbf{n}|=m} \left(\sqrt{\binom{m}{\mathbf{n}}c^{n_0}}\prod_{k=1}^{d} x_k^{n_k}\right)\left(\sqrt{\binom{m}{\mathbf{n}}c^{n_0}}\prod_{k=1}^{d} y_k^{n_k}\right) && (4.25)\\
&= \phi(\mathbf{x})^T\phi(\mathbf{y}) && (4.26)
\end{aligned}$$

□

Example 4.2.2 (Inhomogeneous Quadratic Kernel ($m = d = 2$))
Let $\phi : \mathbb{R}^2 \to \mathbb{R}^6$ be defined as:

$$\phi(\mathbf{x}) = \phi([x_1, x_2]) \coloneqq [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2]^T = \tilde{\mathbf{x}} \quad (4.27)$$

which maps a $d' = 2$ dimensional feature vector to a $d' = 6$ dimensional feature vector . Then given 2 feature vectors $\mathbf{x_i}, \mathbf{x_j}$ we can compute the kernel function in 2 ways:

$$\begin{aligned}
\mathrm{k}_2(\mathbf{x_i}, \mathbf{x_j}) &\coloneqq \phi(\mathbf{x_i})^T\phi(\mathbf{x_j}) && (4.28)\\
&= 1 + 2x_{i,1}x_{j,1} + 2x_{i,2}x_{j,2} + 2x_{i,1}^2x_{j,1}^2 + && (4.29)\\
&\quad x_{i,2}^2x_{j,2}^2 + 2x_{i,1}x_{i,2}x_{j,1}x_{j,2} && (4.30)\\
&= (1 + x_{i,1}x_{j,1} + x_{i,2}x_{j,2})^2 && (4.31)\\
&= (1 + \mathbf{x_i}^T\mathbf{x_j})^2 && (4.32)
\end{aligned}$$

Here, again, we see that the number of operations if we use Lemma 4.2.2 directly is reduced dramatically.

We have reduced a dot product between two vectors $\phi(\mathbf{x})$ of size d' (order $\mathcal{O}((d')^m)$) to one single dot product of two vectors of size d (order $\mathcal{O}(d^m)$). Also remember that we never have to compute $\phi(\mathbf{x})$ in any way, it's implicitly computed by the kernel. Complicated functions like Equation 4.27 must not be derived manually and the computational complexity between homogeneous and inhomogeneous kernels is the same.

4.3 Kernelized Perceptron

We can use the dual optimization problem and the kernel trick to solve efficiently the perceptron algorithm training phase.

Algorithm 4.1: Kernelized Perceptron

```
1 α_0 ← 0
2 foreach t = 1, 2, ..., T
3     sample (x_i, y_i) ∈_{u.a.r.} D_train   ▷ Sample uniformly at random with replacement.
4     if y_i Σ_{j=1}^{n} α_{t,j} y_j k(x_i, x_j) ≥ 0   ▷ Predict using the dual.
5         α_{t+1} ← α_t   ▷ Correct prediction, no update.
6     else
```

```
        α_{t+1} ← α_t
        α_{t+1,i} ← α_{t+1,i} + η_t                ▷ Wrong prediction, update.
    end
    return α_{T+1}
```

Then if we are given a new point $\mathbf{x}$ to predict using the trained preceptron we just check the sign.

$$\hat{y} = \text{sign}\left(\sum_{j=1}^{n} \alpha_j y_j \, \text{k}(\mathbf{x}, \mathbf{x_j})\right) \tag{4.33}$$

4.4 Kernel Properties

Instead of computing the kernel for each combination of inputs $\mathbf{x_1}, \ldots, \mathbf{x_n}$ as a function, we can store all the kernels in a *kernel matrix*.

Definition 4.4.1 (Kernel Matrix) *Let* $\text{k} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ *be a kernel function then the associated* kernel matrix $\mathbf{K}$ *is defined as:*

$$K_{i,j} := \text{k}(\mathbf{x_i}, \mathbf{x_j}) \tag{4.34}$$

The advantage of using a kernel matrix in our model is that once we have computed $\mathbf{K}$ we don't have to store our data $\mathbf{x_1}, \ldots, \mathbf{x_n}$ anymore since it's implicitly contained in $\mathbf{K}$. The kernel has the following properties:

- **Symmetric**:

  Proof.

  $$\begin{aligned} K_{i,j} &:= \text{k}(\mathbf{x_i}, \mathbf{x_j}) && (4.35)\\ &= \phi(\mathbf{x_i})^T \phi(\mathbf{x_j}) && (4.36)\\ &= \phi(\mathbf{x_j})^T \phi(\mathbf{x_i}) && (4.37)\\ &= \text{k}(\mathbf{x_j}, \mathbf{x_i}) = K_{j,i} && (4.38) \end{aligned}$$

  □

- **Positve Semi-definite**:

  Proof.

  $$\begin{aligned} \mathbf{a}^T \mathbf{K} \mathbf{a} &= \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j \, \text{k}(\mathbf{x_i}, \mathbf{x_j}) && (4.39)\\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j \phi(\mathbf{x_i})^T \phi(\mathbf{x_j}) && (4.40)\\ &= \left(\sum_{i=1}^{n} a_i \phi(\mathbf{x_i})\right)^T \left(\sum_{j=1}^{n} a_j \phi(\mathbf{x_j})\right) && (4.41)\\ &= \left\| \sum_{i=1}^{n} a_i \phi(\mathbf{x_i}) \right\|^2 \geq 0 && (4.42) \end{aligned}$$

  □

- **Composition rules**:
  Given kernel functions $\mathrm{k}_i : X \times X \to \mathbb{R}$ defined on some data space X, then all of the following are valid kernels:
  - $\mathrm{k}(x, x') = \mathrm{k}_1(x, x') + \mathrm{k}_2(x, x')$
  - $\mathrm{k}(x, x') = \mathrm{k}_1(x, x')\,\mathrm{k}_2(x, x')$
  - $\mathrm{k}(x, x') = c\,\mathrm{k}_1(x, x')$
  - $\mathrm{k}(x, x') = f(\mathrm{k}_1(x, x'))$
  - $\mathrm{k}(z, z') = \mathrm{k}_1(V(z), V(z'))$
  - $\mathrm{k}(x, x') = \sum_{i=1}^{d} \mathrm{k}_i(x_i, x_i')$ for $x \in \mathbb{R}^d$ *ANOVA Kernel*.

  Where $c \in \mathbb{R}$, f is a polynomial with positive coefficients or the exponential function, and $V : Z \to X$ is some function on the data space.

Lemma 4.4.1 (Feature Map Construction) *Given a finite data space $X = \{1, \ldots, n\}$ i.e. $n < \infty$ and a symmetric positive semi-definite matrix $\mathbf{K} \in \mathbb{R}^{n \times n}$, then we can always construct a feature map $\phi : X \to \mathbb{R}^n$ such that $K_{i,j} = \phi(i)^T \phi(j)$.*

Proof. Since $\mathbf{K}$ is a symmetric positive semi-definite matrix it has real and non negative eigenvalues, and can be decomposed with eigen decomposition as follows:

$$\mathbf{K} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T \tag{4.43}$$

Where $\mathbf{U} = [\mathbf{u_1} \cdots \mathbf{u_n}]$ is the orthonormal matrix of eigenvectors and $\mathbf{\Lambda} = \mathrm{diag}([\lambda_1, \ldots, \lambda_n])$ is the diagonal matrix of eigenvalues, both arranged in non-increasing order of eigenvalues $\lambda_1 \geq \cdots \geq \lambda_n \geq 0$. Then we can define $\mathbf{\Lambda} = \mathbf{\Lambda}^{\frac{1}{2}}\mathbf{\Lambda}^{\frac{1}{2}T}$ and we get:

$$\mathbf{K} = \underbrace{\mathbf{U}\mathbf{\Lambda}^{\frac{1}{2}}}_{:=\mathbf{\Phi}^T} \underbrace{\mathbf{\Lambda}^{\frac{1}{2}T}\mathbf{U}^T}_{=\mathbf{\Phi}} \tag{4.44}$$

Where $\mathbf{\Phi}_i = \phi(i)$, and thus it holds that $K_{i,j} = \mathbf{\Phi}_i^T \mathbf{\Phi}_j = \phi(i)^T \phi(j)$. □

Theorem 4.4.2 (Mercer's) *Let X be a compact subset of $\mathbb{R}^d$, and $k : X \times X \to \mathbb{R}$ a kernel function, then k can be expanded into a uniformely convergent series of bounded functions ϕ_i such that:*

$$\mathrm{k}(x, x') = \sum_{i=1}^{\infty} \lambda_i \phi_i(x) \phi_i(x') \tag{4.45}$$

4.5 Infinite Dimensional Kernels

We can define other types of kernel other than polynomial that have an infinite feature space, often such kernels are referred to as non-parametric kernels.

Definition 4.5.1 (Gaussian Kernel) *Given feature vectors $\mathbf{x_i}, \mathbf{x_j} \in \mathbb{R}^d$ and*

a scalar bandwidth $h \in \mathbb{R}$, *the* Gaussian kernel *is defined as:*

$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp\left(\frac{-\left\|\mathbf{x_i} - \mathbf{x_j}\right\|_2^2}{h^2}\right) \tag{4.46}$$

The Gaussian kernel is useful since it obtains a value close to 1 the closer $\mathbf{x_i}$ is to $\mathbf{x_j}$, and the value approaches 0 as they are farther away. In other words, we can measure the similarity between two points $\mathbf{x_i}$ and $\mathbf{x_j}$ using the Gaussian kernel.

With this information, we can construct a *k nearest neighbor* classifier which doesn't need any training and only uses the provided data to classify a new point.

Algorithm 4.2: KNN

1 ***input*** $\mathbf{X} = [\mathbf{x_1} \cdots \mathbf{x_n}]$, $\mathbf{y} = [y_1, \ldots, y_n]^T$, $k \in \mathbb{N}$, $\mathbf{x} \in \mathbb{R}^d$
2 ***output*** $\hat{y} \in \mathbb{R}$
3 ***return*** $\hat{y} = \text{sign}\left(\sum_{i \in \mathcal{N}_k(x)} k(\mathbf{x_i}, \mathbf{x}) y_i\right)$

where $\mathcal{N}_k(x)$ is the set with the k closest neighbor of x and k is a Gaussian or some other similarity measuring kernel. The downside compared to the kernelized perceptron is that this algorithm uses all of the training data for each new point and thus it's very inefficient. Also, the prediction cannot capture global trends but is only depends on close points.

Definition 4.5.2 (Laplacian Kernel) *Given feature vectors* $\mathbf{x_i}, \mathbf{x_j} \in \mathbb{R}^d$ *and a scalar* bandwidth $h \in \mathbb{R}$, *the* Laplacian kernel *is defined as:*

$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp\left(\frac{-\left\|\mathbf{x_i} - \mathbf{x_j}\right\|_1}{h}\right) \tag{4.47}$$

This Kernel is similar to the Gaussian kernel but it uses exponential decay instead of smooth decay.

4.6 Kernelized SVM

By using Lemma 4.1.1 and the kernel trick we can kernelize the SVM algorithm by finding the dual with $\boldsymbol{\alpha}$.

$$\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i \mathbf{w}^T \tilde{\mathbf{x}}_\mathbf{i}\} + \lambda \|\mathbf{w}\|_2^2 \tag{4.48}$$

$$= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, 1 - y_i \left(\sum_{j=1}^{n} \alpha_j y_j \tilde{\mathbf{x}}_\mathbf{j}\right)^T \tilde{\mathbf{x}}_\mathbf{i}\right\} + \lambda \left\|\sum_{j=1}^{n} \alpha_j y_j \tilde{\mathbf{x}}_\mathbf{j}\right\| \qquad \textit{Lemma 4.1.1} \tag{4.49}$$

$$= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, 1 - \sum_{j=1}^{n} \alpha_j y_i y_j (\tilde{\mathbf{x}}_\mathbf{i}{}^T \tilde{\mathbf{x}}_\mathbf{j})\right\} + \lambda \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \tilde{\mathbf{x}}_\mathbf{i}{}^T \tilde{\mathbf{x}}_\mathbf{j} \tag{4.50}$$

$$= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, 1 - \sum_{j=1}^{n} \alpha_j y_i y_j \, \mathrm{k}(\mathbf{x}_\mathbf{i}, \mathbf{x}_\mathbf{j})\right\} + \lambda \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) \tag{4.51}$$

To write the objective in a more compact way we define $\boldsymbol{\alpha} := [\alpha_1, \ldots, \alpha_n]^T$, $\mathbf{D}_y := \mathrm{diag}(y_1, \ldots, y_n)$, $\mathbf{k}_i = [y_1 \, \mathrm{k}(\mathbf{x}_\mathbf{i}, \mathbf{x}_\mathbf{1}), \ldots, y_n \, \mathrm{k}(\mathbf{x}_\mathbf{i}, \mathbf{x}_\mathbf{n})]^T$ and $\mathbf{K}$ is the kernel matrix. Then the more compact kernelized SVM objective dual is:

Learning

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, 1 - y_i \boldsymbol{\alpha}^T \mathbf{k}_i\right\} + \lambda \boldsymbol{\alpha}^T \mathbf{D}_y \mathbf{K} \mathbf{D}_y \boldsymbol{\alpha} \tag{4.52}$$

Prediction

$$\hat{y} = f(\mathbf{x}) = \mathrm{sign}\left(\sum_{j=1}^{n} \alpha_j y_j \, \mathrm{k}(\mathbf{x}, \mathbf{x}_\mathbf{j})\right) \tag{4.53}$$

4.7 Kernelized Linear Regression

By using Lemma 4.1.1 and the kernel trick we can kernelize the linear regression algorithm by finding the dual with $\boldsymbol{\alpha}$.

$$\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (\mathbf{w}^T \tilde{\mathbf{x}}_i - y_i)^2 + \lambda \|\mathbf{w}\|_2^2 \tag{4.54}$$

$$= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \left(\left(\sum_{j=1}^{n} \alpha_j y_j \tilde{\mathbf{x}}_\mathbf{j}\right) \tilde{\mathbf{x}}_i - y_i\right)^2 + \lambda \left\|\sum_{j=1}^{n} \alpha_j y_j \tilde{\mathbf{x}}_\mathbf{j}\right\|_2^2 \qquad \textit{Lemma 4.1.1} \tag{4.55}$$

$$= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_j (\tilde{\mathbf{x}}_j^T \tilde{\mathbf{x}}_i) + \lambda \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \tilde{\mathbf{x}}_\mathbf{i}{}^T \tilde{\mathbf{x}}_\mathbf{j} \tag{4.56}$$

$$= \min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_j \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) + \lambda \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) \tag{4.57}$$

Then the more compact kernelized linear regression objective dual is:

Learning

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{n} \|\boldsymbol{\alpha}\mathbf{K} - \mathbf{y}\|_2^2 + \lambda \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} \tag{4.58}$$

We can also solve this optimization problem using the following closed form:

$$\boldsymbol{\alpha} = (\mathbf{K} + n\lambda \mathbf{I})^{-1} \mathbf{y} \tag{4.59}$$

Prediction

$$\hat{y} = f(\mathbf{x}) = \sum_{i=1}^{n} \hat{\alpha}_i \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}) \tag{4.60}$$

Semi-parametric Regression If we have some periodic data we can use a *e.g.* Gaussian kernel and with the right parameters it will fit the periodicity correctly. However, if the data is both periodic but also follows a linear trend is some direction using only a Gaussian kernel will not be enough. It would be optimal if we could use parametric models that are rigid (*e.g.* linear or polynomial kernels) and very good for general high level trends, but also non-parametric models *e.g.* Gaussian kernels that can handle variability and periodicity. We have seen before that the composition of two kernels is still a valid kernel and thus we can combine more than one single kernel to get semi-parametric regression models.

Example 4.7.1 (Semi-parametric kernel)

$$\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) = c_1 \underbrace{\exp\left(\frac{-\left\|\mathbf{x_i} - \mathbf{x_j}\right\|_2^2}{h^2}\right)}_{\text{Non-parametric kernel (Gaussian)}} + c_2 \underbrace{\mathbf{x}_i^T \mathbf{x}_j}_{\text{Parametric kernel (Linear, } m = 1\text{)}} \tag{4.61}$$

This kernel used on kernelized linear regression will fit both linear and periodic data at the same time.

Kernel Tuning We have seen a lot of different kernels, in semi-parametric regression often the choice of the kernel is not obvious. There are less advanced methods to select the kernel *e.g.* brute force or domain knowledge (*i.e.* like in Example 4.7.1 in which we choose a linear and Gaussian kernel knowing that our data was periodic in some linearly defined direction), but also more advanced algorithms used for kernel learning.

Neural Networks 5

5.1 Introduction 36
5.2 General Neural Network . . 37
5.3 Forward Propagation 39
5.4 Objective 40
5.5 Computational Graphs 41
5.6 Back-Propagation 45
5.7 Weight Initialization 48
5.8 Optimizers 50
5.9 Overfitting 50
5.10 Regularization 51
5.11 Convolutional Neural Networks (CNNs) 53
5.12 Other 58

Neural networks were first idealized in the 1950s. In the 1980s many papers were published about them, however, at some point, SVMs were discovered and neural networks lost popularity. In the 2000s neural networks came back thanks to the advances in computation and the increasing amount of data. Today, neural networks are everywhere as the main component of the field of deep learning.

5.1 Introduction

We have seen that when dealing with non-linearly separable data we can transform the features with non-linear functions (*e.g.* polynomials) and then apply kernels to be able to use non-linear classification or regression. This method has some drawbacks: in facts, choosing the right kernel can be challenging since different applications need different kernels, and with the increasing amount of data also the computational complexity increases. The goal of neural networks is to automatically learn a non-linear mapping ϕ from a labeled dataset.

Goal

Given a set of feature vectors $\mathbf{x_1}, \dots, \mathbf{x_n}$ where $\mathbf{x_i} \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$), and a set labels $\mathbf{y}_1, \dots, \mathbf{y}_n$ where $\mathbf{y}_i \in \mathbb{R}^k$ (which can be represented as a matrix $\mathbf{Y} \in \mathbb{R}^{n \times k}$).

Output the coefficients $\boldsymbol{\theta}$ such that

$$f(\mathbf{x}_i; \boldsymbol{\theta}) = \hat{\mathbf{y}}_i \approx \mathbf{y}_i, \quad \forall i \in \{1, \dots, n\} \tag{5.1}$$

Basic Neural Network We still have to define f, we start with the most basic neural network possible.

$$f(\mathbf{x}_i; \boldsymbol{\theta}, \mathbf{w}) := \sum_{j=1}^{m} w_j \phi(\mathbf{x}; \theta_j) = \phi(\mathbf{x}; \boldsymbol{\theta})^T \mathbf{w} \tag{5.2}$$

Then use this definition of f and try to learn ϕ by minimizing some loss function $\mathcal{L}$:

$$\min_{\mathbf{w}, \boldsymbol{\theta}} \sum_{i=1}^{n} \mathcal{L}\left(\mathbf{y}_i; f(\mathbf{x}_i; \mathbf{w}, \boldsymbol{\theta})\right) \tag{5.3}$$

Where ϕ will be a non-linear *activation function*.

Activation Functions

Definition 5.1.1 (Activation Function) *Let $\boldsymbol{\theta} \in \mathbb{R}^m$ be a vector of learnable parameters for some function ϕ and $\mathbf{x} \in \mathbb{R}^m$ be the input vector, then:*

$$\phi(\mathbf{x}; \boldsymbol{\theta}) := \varphi(\boldsymbol{\theta}^T \mathbf{x}) \tag{5.4}$$

where $\varphi : \mathbb{R} \to \mathbb{R}$ is called activation-function.[a]

[a] Often we only write ϕ and refer to it as activation function, however the actual activation function is φ which takes a scalar as input.

Note that in the study of neural networks we often give up the notion of a convex non-linear function for non-convex non-linear activation function, where the optimal convergence is no longer guaranteed. We will argue later on that this is not a big problem for training.

Let $z := \boldsymbol{\theta}^T \mathbf{x}$, then the most used activation functions are:

- **Sigmoid**

$$\varphi(z) = \frac{1}{1 + \exp(-z)} \tag{5.5}$$

$$\varphi'(z) = \varphi(z)(1 - \varphi(z)) \tag{5.6}$$

- **Tanh**

$$\varphi(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \tag{5.7}$$

$$\varphi'(z) = 1 - \varphi(z)^2 \tag{5.8}$$

- **ReLU**

$$\varphi(z) = \max(z, 0) \tag{5.9}$$

$$\varphi'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \textit{undefined} & \text{if } z = 0 \end{cases} \tag{5.10}$$

5.2 General Neural Network

In the basic neural network, we used a single non-linear activation function which can be used to approximate simple non-linear functions f. However, in the case that fitting our dataset requires a more complex function the basic setting will not be enough. The general idea is that instead of giving $\mathbf{x}$ as input to the activation function we can nest many activation functions by continuously giving the output of the previous activation function as input to the next one. To understand this idea we can use different notations, the standard mathematical notation gives us a precise understanding of how to evaluate the different activation functions but is often harder to have an intuitive grasp of what is going on. The graphical notation is much more intuitive but might miss some details that only the mathematical one provides. Since different notation give us different understandings we will provide many of them.

Mathematical View Differently form the basic setting, here we use many nested functions. Each function has its weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ (in the basic settings we only had a coefficient matrix $\boldsymbol{\theta}$). We will have many layers of parameters $\boldsymbol{\theta} \coloneqq (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)})$, then to compute f:

$$f(\mathbf{x}; \boldsymbol{\theta}) = f^{(L)}(\cdots f^{(2)}(f^{(1)}(\mathbf{x}; \boldsymbol{\theta}); \boldsymbol{\theta}) \cdots ; \boldsymbol{\theta}) \tag{5.11}$$

Then for each *layer* $l \in \{1, \dots, L\}$ the function $f^{(l)}$ uses only it own weights, biases, and possibly a different activation function $\varphi^{(l)}$. Then each intermediate vector will be called a *hidden layer* $\mathbf{h}^{(l)}$.

$$\mathbf{h}^{(1)} \coloneqq f^{(1)}(\mathbf{x}; \mathbf{W}^{(1)}, \mathbf{b}^{(1)}) = \varphi^{(1)}\left(\mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right) \tag{5.12}$$

$$\mathbf{h}^{(2)} \coloneqq f^{(2)}(\mathbf{h}_1; \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) = \varphi^{(2)}\left(\mathbf{h}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}\right) \tag{5.13}$$

$$\vdots$$

$$\mathbf{h}^{(L)} \coloneqq f^{(L)}(\mathbf{h}^{(L-1)}; \mathbf{W}^{(L)}, \mathbf{b}^{(L)}) = \varphi^{(L)}\left(\mathbf{h}^{(L-1)}\mathbf{W}^{(L)} + \mathbf{b}^{(L)}\right) \tag{5.14}$$

Where $\mathbf{h}^{(0)} \coloneqq \mathbf{x}$, is the *input* layer and $\mathbf{h}^{(L)} \coloneqq f(\mathbf{x}; \boldsymbol{\theta}) = \hat{\mathbf{y}} \approx \mathbf{y}$, is the *output* layer, however we usually use keep the notation $\mathbf{x}$ and $\hat{\mathbf{y}}$ to show that each function can be seen as taking a hidden layer as input and returning a hidden layer as output. The dimensions are $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$, $\mathbf{b}^{(l)} \in \mathbb{R}^{1 \times d^{(l)}}$, where $d^{(l)}$ is the number of hidden units in layer l, *i.e.* $\mathbf{h}^{(l)} \in \mathbb{R}^{d^{(l)}}$, and we use the convention that $d^{(0)} \coloneqq d$ and $d^{(L)} \coloneqq k$.

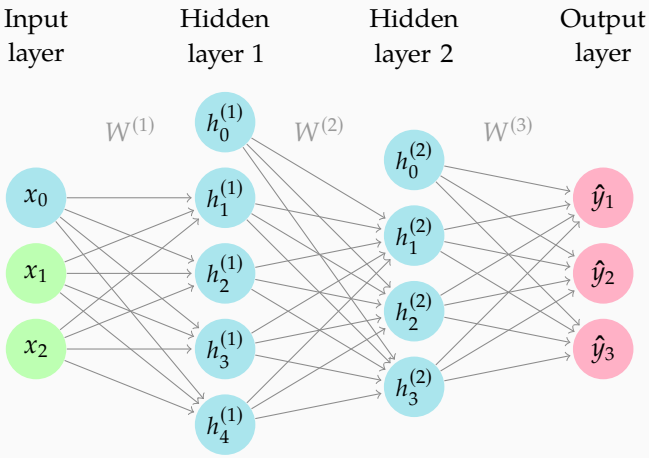
Note that depending on the way we build the neural network each hidden layer (and thus weight matrix $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$) can have arbitrary size, the only constraint is in the input and output layers that are fixed by the dimensionality of our dataset. The number of layers L can be chosen depending on how complex is the function f that we want to approximate. All of those sizes must be chosen manually and are called *hyper-parameters* of the network, there is no general rule of thumb that works for all functions f.

Graph View We can visualize the previous mathematical view in a network graph view to get a more intuitive understanding. Here, instead of viewing each layer as a vector operation on the previous layer, we can view each element of the hidden vector as a *neuron*. The value of the neuron is evaluated as a weighted sum of the previous layer neurons and the strength of each connecting edge weight. More formally, assume that the hidden layer l has dimension $d^{(l)} \in \mathbb{N}$, then:

$$h_j^{(l)} = \varphi^{(l)}\left(b_j^{(l)} + \sum_{i=1}^{d^{(l)}} h_i^{(l-1)} W_{i,j}^{(l)}\right) \tag{5.15}$$

Neural networks are similar to the perceptron algorithm, instead, here, we use a non-linear activation function, and thus each neuron returns a real-valued output (not only -1 or 1) to learn non-linear features. Instead of a single layer we have many layers, each layer can learn more complicated features by taking all outputs of the previous neurons (which in reality are just number in a vector) and combining them with a weighted sum to which we apply a non-linear function.

Example 5.2.1 (Neural Network Graph View) In this example we input dimension $d = 2$, output dimension $k = 3$, then we choose to use $L = 2$ hidden layers where $d^{(1)} = 4$ and $d^{(2)} = 3$. Remember that the input layer can also be seen as the hidden layer 0, $x_j = h_j^{(0)}$, and the output layer can also be seen as the hidden layer 3, $\hat{y}_j = h_j^{(3)}$. Furthermore to simplify the evaluation for the bias we let, $h_0^{(l-1)} = 1$ and the weights $W_{0,j}^{(l)} = b_j^{(l)}$, such that the first row of the weight matrix of layer l will contain all the biases for layer l, and we won't need to add the biases separately. Then we can represent $f(\mathbf{x}; \boldsymbol{\theta})$ using the following graph:



Then for example: $h_1^{(1)} = \varphi^{(1)}\big(\underbrace{x_0 W_{0,1}^{(1)}}_{=1 \cdot b_1^{(1)}} + x_1 W_{1,1}^{(1)} + x_2 W_{2,1}^{(1)}\big)$.

Usually, we never compute the sum presented in the graph view but use the mathematical view with matrix multiplication and thus each layer is computed in one go. The graph view is just another way to see the mathematical view *i.e.* the output is the same, however instead of vector operations we can visualize how the input $\mathbf{x}$ flows to the output $\hat{\mathbf{y}}$, this process is called *forward propagation*.

5.3 Forward Propagation

The previous different views of a neural network give us ways to understand the process of forward propagation more intuitively. One nice functionality of neural networks is that by selecting appropriately the output layer function we can solve both regression or classification problems.

Algorithm 5.1: NN Forward Propagation

$\mathbf{h}_0 \leftarrow \mathbf{x}$
foreach $l = 1 : L$
    $\mathbf{z}^{(l)} = \mathbf{h}^{(l-1)}\mathbf{W}^{(l)}$
    $\mathbf{h}^{(l)} = \varphi^{(l)}(\mathbf{z}^{(l)})$
end
return $h^{(L)}$

If we want to solve a regression problem, the function $\varphi^{(L)}$, will be the identity function $\varphi^{(L)}(\mathbf{z}^{(L)}) = \mathbf{z}^{(L)}$. For classification problems if the output dimension is $k = 1$ then $\varphi^{(L)}(\mathbf{z}^{(L)}) = \text{sign}(\mathbf{z}^{(L)})$, if instead we are dealing with multi-class classification *i.e.* $k > 1$, we will use $\varphi^{(L)}(\mathbf{z}^{(L)}) = \arg\max_i(\mathbf{z}^{(L)})_i$ to select the class with the highest score.

As we have seen before there is no general rule that tells us how many hidden layers we should use, however the *universal approximation theorem* tells us that

Theorem 5.3.1 (Universal Approximation) *Let $\varphi(\mathbf{z})$ be the Sigmoid activation function and $f^*(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^k$ a continuous function $f \in C^{(1)}$, then there always exists a finite ($m < \infty$) sum of the form*

$$f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) = \sum_{j=1}^{m} w_j \varphi(\boldsymbol{\theta}^T \mathbf{x}) \tag{5.16}$$

such that for all $\varepsilon > 0$

$$\left| f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) - f^*(\mathbf{x}) \right| < \varepsilon \quad \forall \mathbf{x} \tag{5.17}$$

More simply, the basic neural network with a single layer $L = 1$ and the right activation function could be enough to approximate *any* continuous function with an error as small as we want. If this is true, why we might want more than a single layer? The reason is that m might be really large and is often unknown. A nice property of multi-layer neural networks is that by adding only a few layers we can exponentially decrease the size of m.

5.4 Objective

Forward propagation is used for prediction and thus assumes that we already have the parameters $\boldsymbol{\theta}$. Stochastic gradient descent is used to train the network and thus find the correct parameters $\boldsymbol{\theta}$ such that $f(\mathbf{x}; \boldsymbol{\theta}) = \hat{\mathbf{y}} \approx \mathbf{y}$ for any pair $(\mathbf{x}, \mathbf{y})$ in the dataset.

Recall that if we add an additional input at each hidden layer with value 1, we can merge the biases as the first row in each layer's weight matrix $\mathbf{W}^{(l)}$. This will allow us to represent our parameters in a more compact form $\boldsymbol{\theta} := (\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(L)})$.

We can then define the objective function of a neural network as

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta} \in \mathbb{R}^D} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i) \tag{5.18}$$

where $\mathcal{L}$ is a *multi-output* loss function.

Definition 5.4.1 (Multi-output loss) *Given a standard loss function $\ell_\star(y, \hat{y})$, $\ell_\star : \mathbb{R}^2 \to \mathbb{R}$, vectors $\boldsymbol{\theta}$,$\mathbf{x}$, and $\mathbf{y}$, the* multi-output loss

$\mathcal{L} : \mathbb{R}^D \to \mathbb{R}$ *is defined as:*

$$\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) := \frac{1}{k} \sum_{j=1}^{k} \ell_\star(y_j, f(\boldsymbol{\theta}; \mathbf{x})_j) \tag{5.19}$$

More simply it's just a function that averages the standard loss of each component of the output vector if we are using a neural network with many outputs. When we are dealing with regression $\star$ is usually a mean squared error, and if we are dealing with classification a multi-class perceptron or hinge loss.

As we have seen the when computing the forward propagation for f, we apply non-linear and possibly non-convex activation functions. The reason we do this that the advantages outweigh the disadvantages. In fact, we can still get a very good approximation of the optimal solution for $\boldsymbol{\theta}$ by using stochastic gradient descent even if the problem is non-convex.

SGD Update Rule

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\boldsymbol{\theta}_t; \mathbf{x}, \mathbf{y}) \tag{5.20}$$

5.5 Computational Graphs

In the previous models, when we had to compute the gradient of the loss function, it was more or less straight forward. In this case, the gradient of the loss $\mathcal{L}$ with respect to $\boldsymbol{\theta}$ contains the output $f(\boldsymbol{\theta}, \mathbf{x})$ of the forward propagation, thus it can be very tedious to compute. We will introduce the idea of a *computational graph*, which is a way to visualize the computation of gradients of any multivariate function, then show how this process can be easily extended to compute the gradient of our specific loss function very efficiently. This process is not only a way to visualize how to compute partial derivatives, but also the method that most machine learning software use to compute complicated derivatives, called *automatic differentiation*.

Given any multivariate function $F(x_1, \ldots, x_n)$, to compute the partial derivative $\frac{\partial F}{\partial x_i}$, we will first build a computational graph, and then read the output partial derivative from the graph. More precisely we will follow the next steps, later we will show a concrete example.

Build Graph First, attribute a variable to the output of each sub-function of F. Secondly, each input of F will be a node without any edges pointing at it, ad each output of F will be a node that doesn't point to any other node, and each intermediate variable that is associated with the output of a sub-function of F will also be a node. Nodes will have edges pointing at them from other nodes that are input, and will point to nodes where their output is passed to. Since each node represents a variable once we will have computed F each node will contain a numerical value. Then if we have a *direct* connection from a source node with variable s to node with target node with variable t, the intermediate edge will represents the partial derivative $\frac{\partial t}{\partial s}$.

Compute Partial Derivative There are three ways to compute partial derivatives depending on the structure of the graph, each with its own tradeoffs.

1. *Standard Chain Rule*, has the advantage that it lets us compute the derivative between *any* two nodes in the graph s and t that are *not* directly connected *i.e.* $\frac{\partial t}{\partial s}$. To do so we will follow the path in the graph from s to t and multiply the value of the derivative of each intermediate edge, if there are multiple paths from s to t we will sum their values.
2. *Forward-Mode Differentiation*, lets us compute the derivative of any node with respect to *one* input, *i.e.* apply the operator $\frac{\partial}{\partial x_i}$ to each node. In this case, instead of working with edges, we will save in each intermediate node n the derivative $\frac{\partial n}{\partial x_i}$. We will start by setting the derivative of $\frac{\partial x_j}{\partial x_i}$ to 0 if $i \neq j$ and to 1 if $i = j$. Then we will follow the graph from the inputs to the outputs and each intermediate node n will sum the value of its children's partial derivatives multiplied by the derivative of the connecting edge and flow them forward as output. This process will be repeated until we have all derivatives with respect to each node, included $\frac{\partial F_j}{\partial}$.
3. *Backward-Mode Differentiation*, lets us compute the derivative of *one* output node with respect to all other nodes, *i.e.* apply the operator $\frac{\partial F_j}{\partial}$ to each node. The process is similar to forward mode differentiation but instead flows from outputs to inputs. We will set $\frac{\partial F_j}{F_i}$ to 0 if $i \neq j$ and to 1 if $i = j$, then store in each intermediate node n the derivative $\frac{\partial F_j}{\partial n}$, to do so we will sum all parents' partial derivatives multiplied by the derivative of the connecting edge and flow them backward.

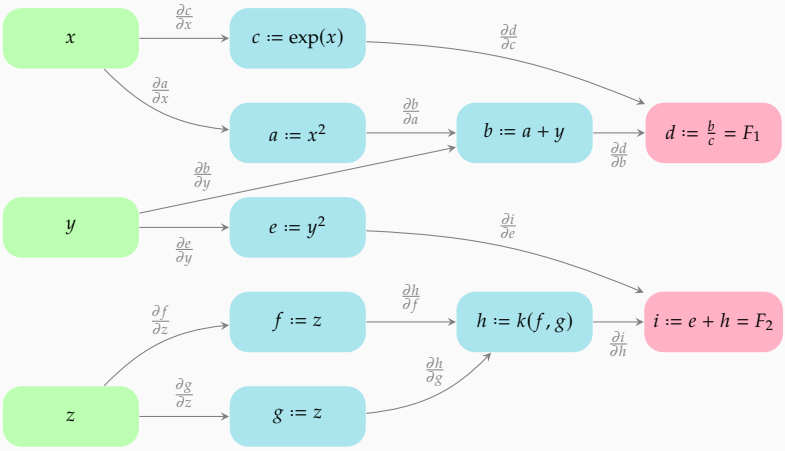
One obvious question is, why we need three different methods if the first one works for all possible partial derivatives? The reason is that following all paths between two nodes as in standard chain rule might lead to an exponential explosion. When using forward or backward mode differentiation we consider for each node only the children or parent nodes and thus it's much more efficient. Furthermore, depending on the function it might be more efficient to use forward-mode differentiation if the number of output of F is much larger than the number of outputs, conversely, backward-mode, differentiation is much more efficient if the number of inputs is much larger than the number of outputs. [1]

Example 5.5.1 (Standard Chain Rule) Let $F : \mathbb{R}^3 \to \mathbb{R}^2$, be defined as:

$$F(x, y, z) := \begin{bmatrix} \frac{x^2+y}{\exp(x)} & y^2 + k(z, z) \end{bmatrix}^T = \begin{bmatrix} F_1 & F_2 \end{bmatrix}^T \tag{5.21}$$

where $k : \mathbb{R}^2 \to \mathbb{R}$ can be any function (used to demonstrate how the graph looks if we have undefined functions), then we start by building the graph with one variable that represents a single sub-function per node, where the edges represent the derivative of the successive node with respect to the previous one. Green nodes will represent inputs, blue nodes sub-functions, and red nodes outputs.

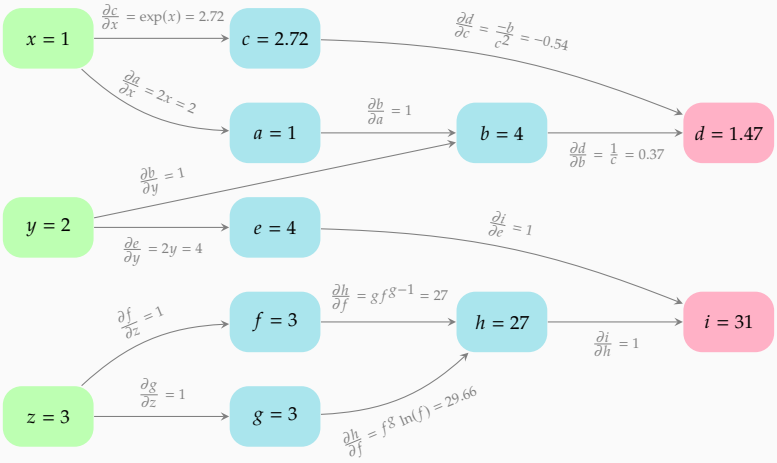
1: Computational graphs and the neural network graph view are *not* the same thing. In the computational graph each variable is represented by a node and the edges represent partial derivatives, wherein the neural network graph each node represents a neuron output and each edge a connection weight. Neural network graphs are used to get an intuitive understanding of how the network works but not implemented directly. The computational graph also gives an intuitive understanding but are actually implemented by many libraries to compute first-order derivatives with automatic differentiation.



Now if we want to compute the partial derivative of F_1 with respect to x, $\frac{\partial F_1}{\partial x}$, using the standard chain rule, we will simply sum all the paths from node d to node x (or x to d) in the graph and multiply their partial derivatives.

$$\frac{\partial F_1}{\partial x} = \frac{\partial d}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial x} + \frac{\partial d}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x} \tag{5.22}$$

More concretely let's see a numerical example, let $(x, y, z) = (1, 2, 3)$ and define $k(f, g) := f^g$.



Then again by following the two paths from d to x, the previous partial derivative is:

$$\frac{\partial F_1}{\partial x} = \frac{\partial d}{\partial x} = -0.54 \cdot 2.72 + 0.37 \cdot 1 \cdot 2 = -0.73 \tag{5.23}$$

Or in more simple terms, if we change the value of x by 1, the value of d (which is the output of F_1) changes by approximately -0.73, this idea is really important because it shows us how by changing one variable in the graph (which in this case is the input, but could be any variable) affects the change to another variable.

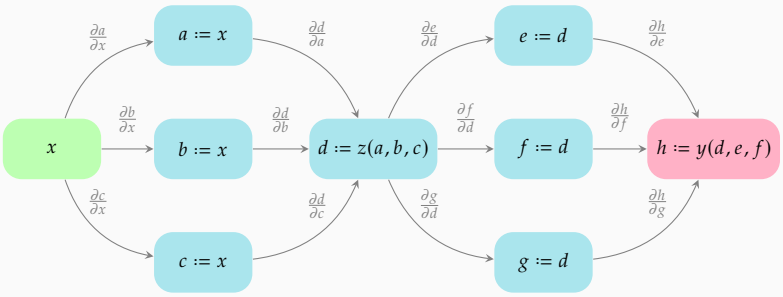
Example 5.5.1 shows us how we can use the graph to compute the partial derivative of any two variables, however, as we have seen before there are some cases in which the number of paths between two variables can increase exponentially, and thus the standard chain rule is very

inefficient.

Example 5.5.2 (Backward-Mode Differentiation) Let $F : \mathbb{R} \to \mathbb{R}$, be defined as:

$$F(x) \coloneqq y(z(x,x,x), z(x,x,x), z(x,x,x)) \tag{5.24}$$

for some functions $y, z : \mathbb{R}^3 \to \mathbb{R}$, note that usually we would write F as $F(x) \coloneqq y(d,d,d)$ where $d \coloneqq z(x,x,x)$.



This time if we want to compute the derivative $\frac{\partial F}{\partial x} = \frac{\partial h}{\partial x}$ the number of paths from h to x is $3^2 = 9$.
Standard chain rule:

$$\frac{\partial h}{\partial x} = \underbrace{\frac{\partial h}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial a}\frac{\partial a}{\partial x}}_{\text{Path 1}} + \cdots + \underbrace{\frac{\partial h}{\partial g}\frac{\partial g}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial x}}_{\text{Path 9}} \tag{5.25}$$

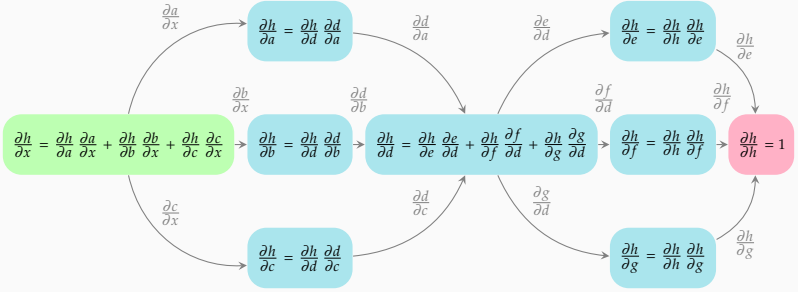
For each path we compute 4 derivatives for a total of $9 \cdot 4 = 36$ derivatives.
Bachward-mode idea:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial d} \cdot \frac{\partial d}{\partial x} \tag{5.26}$$

$$= \left(\frac{\partial h}{\partial e}\frac{\partial e}{\partial d} + \frac{\partial h}{\partial f}\frac{\partial f}{\partial d} + \frac{\partial h}{\partial g}\frac{\partial g}{\partial d}\right) \cdot \left(\frac{\partial d}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial d}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial d}{\partial c}\frac{\partial c}{\partial x}\right) \tag{5.27}$$

In this case we only compute 12 derivatives. It's easy to see that if the depth of this kind of function increases, the number of derivatives with the standard chain rule increases exponentially. With backward-mode differentiation the number of derivatives its linear in both depth and width of the graph.

Backward-mode differentiation starts from h and flows backward through the directed graph, each time storing in the node the derivative with respect to $F = h$, *i.e.* applying the operator $\frac{\partial h}{\partial}$ on all nodes. Thus we start from the node h and store its derivative $\frac{\partial h}{\partial h} = 1$ (if the function has multiple outputs note that $\frac{\partial h}{\partial F} = 0$), then flow backward through the graph where each node will add all derivatives contained on their parent nodes multiplied by the edge from which they came from.



Other than avoiding an exponential explosion, backward-mode differentiation computes the derivative of one output with respect to *all* variables, instead of just one as with standard chain rule. This is really important, each node tells us how much influence it has on the output. For example, if we observe that $\frac{\partial h}{\partial a} = 3$ this tells us that if we would change a by 1, then the output of F would change by approximately 3. If our goal is to make F as small or as big as possible we can just check the graph and see which variables are responsible for the greatest increase or decrease if a small change is applied. Note that if we are dealing with multivariate functions backward-mode differentiation will compute all derivatives, including the ones in the gradient $\nabla_{\mathbf{x}} F$, which is just the vector of the derivatives of one output (red) with respect to all inputs (green nodes).

All of the previous methods are useful to compute complicated derivatives, but how does this apply to neural networks? In fact, neural networks are just complicated functions with some variables ($\boldsymbol{\theta}$) that can be changed as needed. Our goal is to optimize $\mathcal{L} : \mathbb{R}^D \to \mathbb{R}$, *i.e.* a function with a lot of inputs and a single output. We have seen before that for functions with few outputs and many inputs, backward-mode differentiation is the most suitable way to compute derivatives with respect to any variable. This process of computing the derivatives of a loss function ($\nabla_{\boldsymbol{\theta}} \mathcal{L}$) for a neural network is called *back-propagation*. Note that the loss function $\mathcal{L}$ will always have a one-dimensional output even if the neural network has a very high dimensional output.

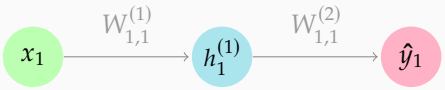
5.6 Back-Propagation

In back-propagation the goal is to compute $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \mathbf{x}, \mathbf{y})$. Recall that the gradient of $\mathcal{L}$ with respect to $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)})$ is the vector of all partial derivatives $\frac{\partial \mathcal{L}}{\partial W_{i,j}^{(l)}}$, $\forall i, j, l$.

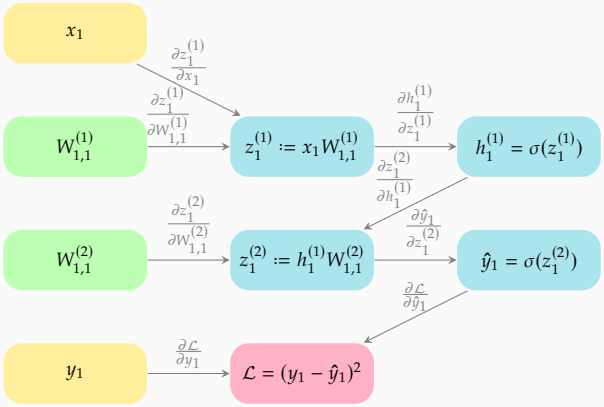
Example 5.6.1 In this example we will compute back-propagation for the following single hidden layer neural network with one input, one output, and no biases (for semplicity).

$$f(x_1; \boldsymbol{\theta}) = \varphi^{(2)}(\varphi^{(1)}(x_1 W_{1,1}^{(1)}) W_{1,1}^{(2)}) = \hat{y}_1 \tag{5.28}$$

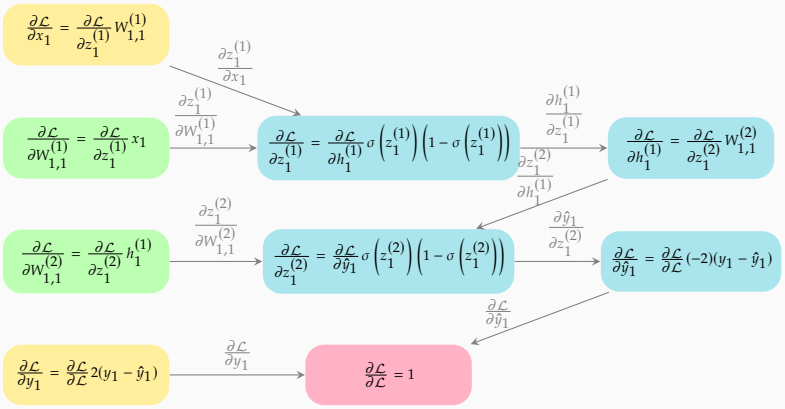
where $\boldsymbol{\theta} = (W_{1,1}^{(1)}, W_{1,1}^{(2)})$, and f has the following graph view.



As we have seen before we will use a computational graph for the loss $\mathcal{L}(\boldsymbol{\theta}; x_1, y_1)$, where green nodes are inputs (*i.e.* $\boldsymbol{\theta}$), blue nodes are sub-functions, red nodes are outputs (*i.e.* $\mathcal{L}$) and *yellow* nodes are fixed variables (*i.e.* x_1, y_1). Note that when computing the loss we want to find the partial derivatives with respect to the weights in $\boldsymbol{\theta}$, and thus x_1,y_1 are not inputs as in f but fixed variables. We will assume that $\varphi^{(1)}$, $\varphi^{(2)}$ are sigmoid activation functions σ, and that our single output loss is the mean squared error. Then the computational graph of $\mathcal{L}$ is:



Then we will apply backward-mode differentiation from the loss function, where the goal is to find the derivatives $\frac{\partial \mathcal{L}}{\partial W^{(1)}_{1,1}}$ and $\frac{\partial \mathcal{L}}{\partial W^{(2)}_{1,1}}$.



Then we can easily read out our derivatives from the graph. Note that for for spacing reasons we wrote the derivatives in a more compact way, usually we just have to read out the value of the partial derivatives from the green nodes. Also the yellow nodes are never actually computed since we can't change our data, however they are just variables and thus nothing is stopping us from computing their partial derivatives.

In this case the extended form is:

$$\frac{\partial \mathcal{L}}{\partial W_{1,1}^{(1)}} = \frac{\partial \mathcal{L}}{\partial z_1^{(1)}} x_1 \tag{5.29}$$

$$= \frac{\partial \mathcal{L}}{\partial h_1^{(1)}} \sigma\left(z_1^{(1)}\right)\left(1 - \sigma\left(z_1^{(1)}\right)\right) x_1 \tag{5.30}$$

$$= \frac{\partial \mathcal{L}}{\partial z_1^{(2)}} W_{1,1}^{(2)} \sigma\left(z_1^{(1)}\right)\left(1 - \sigma\left(z_1^{(1)}\right)\right) x_1 \tag{5.31}$$

$$= \frac{\partial \mathcal{L}}{\partial \hat{y}_1} \sigma\left(z_1^{(2)}\right)\left(1 - \sigma\left(z_1^{(2)}\right)\right) W_{1,1}^{(2)} \sigma\left(z_1^{(1)}\right)\left(1 - \sigma\left(z_1^{(1)}\right)\right) x_1 \tag{5.32}$$

$$= (-2)(y_1 - \hat{y}_1)\sigma\left(z_1^{(2)}\right)\left(1 - \sigma\left(z_1^{(2)}\right)\right) W_{1,1}^{(2)} \sigma\left(z_1^{(1)}\right)\left(1 - \sigma\left(z_1^{(1)}\right)\right) x_1 \tag{5.33}$$

$$\frac{\partial \mathcal{L}}{\partial W_{1,1}^{(2)}} = \frac{\partial \mathcal{L}}{\partial z_1^{(2)}} h_1^{(1)} \tag{5.34}$$

$$= \frac{\partial \mathcal{L}}{\partial \hat{y}_1} \sigma\left(z_1^{(2)}\right)\left(1 - \sigma\left(z_1^{(2)}\right)\right) h_1^{(1)} \tag{5.35}$$

$$= (-2)(y_1 - \hat{y}_1)\sigma\left(z_1^{(2)}\right)\left(1 - \sigma\left(z_1^{(2)}\right)\right) h_1^{(1)} \tag{5.36}$$

The last thing to realize is that we don't have to compute most of the values in the partial derivatives. For example $\sigma\left(z_1^{(1)}\right)$ and $\sigma\left(z_1^{(2)}\right)$ are computed during forward propagation together with $h_1^{(1)}$, $h_1^{(2)}$, and $\hat{y}_1$. Furthermore, since we use backward-mode differentiation, $\frac{\partial \mathcal{L}}{\partial z_1^{(2)}}$ is computed only once and used to evaluate both $\frac{\partial \mathcal{L}}{\partial W_{1,1}^{(1)}}$ and $\frac{\partial \mathcal{L}}{\partial W_{1,1}^{(2)}}$. After having computed the partial derivatives of the loss with respect to the weights $\boldsymbol{\theta}$, we will know by how much the loss changes if we slightly change the weights of the neural network. This information will be used by gradient descent to take small steps towards the optimal value for $\mathcal{L}$.

Example 5.6.1 shows how to compute the partial derivatives of a simple neural network, where the advantages of backward-mode differentiation are not fully expressed as we only have one dimensional layers. In the following example we will show how backward-mode differentiation can be used with wider networks *i.e.* with matrices.

Example 5.6.2

The architecture of standard neural networks is always the same (only the number of layers, the width of each layer, the activation functions, or the single output loss are changing) thus we don't have to build the computational graph for every new standard neural network but only run the following iterative algorithm.

Algorithm 5.2: Standard back-propagation

1 $\nabla_{\mathbf{z}^{(L)}} \mathcal{L} \coloneqq \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \hat{y}_1} & \frac{\partial \mathcal{L}}{\partial \hat{y}_2} & \cdots & \frac{\partial \mathcal{L}}{\partial \hat{y}_k} \end{bmatrix}$ ▷ *Compute output error gradient.*

2 $\nabla_{\mathbf{W}^{(L)}} \mathcal{L} \coloneqq \mathbf{h}^{(L-1)} \cdot \nabla_{\mathbf{z}^{(L)}} \mathcal{L}$ ▷ *Output weight error gradient.*

3 ***foreach*** $l = L - 1 : 1$

4 $\quad \nabla_{\mathbf{z}^{(l)}} \mathcal{L} \coloneqq \varphi'^{(l)}(\mathbf{z}^{(l)}) \odot \left(\nabla_{\mathbf{z}^{(l+1)}} \mathcal{L} \cdot \mathbf{W}^{(l+1)}\right)$ ▷ *Hidden layer l error gradient.*

$\nabla_{\mathbf{W}^{(l)}}\mathcal{L} \coloneqq \mathbf{h}^{(l)}\nabla_{\mathbf{z}^{(l)}}\mathcal{L}$ ▷ *Weight layer l error gradient.*
end
return $\nabla_{\boldsymbol{\theta}}\mathcal{L} = \begin{bmatrix}\nabla_{\mathbf{W}^{(1)}}\mathcal{L} & \nabla_{\mathbf{W}^{(2)}}\mathcal{L} & \cdots & \nabla_{\mathbf{W}^{(L)}}\mathcal{L}\end{bmatrix}$

Note that if we want to compute the gradient of the loss of a different architecture (*e.g.* RNN, LSTM, GRU), and thus a different function, we would have to build a new computational graph and find a new iterative algorithm. However, using backward-mode differentiation we can easily compute the gradient for any network or function. If we are using a library that implements automatic differentiation this process will be automated.

Recap The steps of the training process of a neural network are:

1. Initialize the parameters $\boldsymbol{\theta}$.
2. Choose an optimization algorithm *e.g.* gradient descent.
3. Choose a single output loss function $\ell_\star$.
4. Repeat the following steps:
   a) Pick an input output pair $(\mathbf{x}_i, \mathbf{y}_i)$ from the dataset (or mini-batch pair $(\mathbf{X}_{1\dots B} = [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_B}]^T, \mathbf{Y}_{1\dots B} = [\mathbf{y}_{i_1}, \dots, \mathbf{y}_{i_B}]^T)$).
   b) Forward propagate the pair: $f(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}) = \hat{\mathbf{y}}_i$.
   c) Compute the gradient of the loss function $\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i)$ with respect to the weights $\boldsymbol{\theta}$ using back-propagation.
   d) Use the optimization algorithm to update the parameters $\boldsymbol{\theta}$.

5.7 Weight Initialization

We have seen that the first step to train a neural network model is to initialize the weights $\boldsymbol{\theta}$ in some way. We will show that the initialization of the parameters is really important, and that if it's not properly applied the network won't be able to learn. Recall the two formulas applied during forward and back-propagation:

Forward Propagation Step

$$h_j^{(l)} = \varphi^{(l)}\underbrace{\left(\sum_{i=1}^{d^{(l)}} h_i^{(l-1)} W_{i,j}^{(l)}\right)}_{z_j^{(l)}} \tag{5.37}$$

Back-Propagation Step

$$\frac{\partial \mathcal{L}}{\partial W_{i,j}^{(l-1)}} = h_i^{(l-1)} \varphi'^{(l-1)}(z_i^{(l-1)}) \frac{\partial \mathcal{L}}{\partial h_j^{(l)}} \tag{5.38}$$

During back-propagation we multiply iteratively the derivative of the activation function φ at $z_i^{(l-1)}$. If the variance of $z_i^{(l-1)}$ is very high (*i.e.* $\mathbb{V}[z_i^{(l-1)}] \gg 1$) and we are using a Sigmoid/TanH activation function, its derivative will be always close to 0. Conversely if the variance is very small (*i.e.* $\mathbb{V}[z_i^{(l-1)}] \ll 1$) the derivative of the activation function will always be around the same value. The first problem is called *vanishing*

gradients problem, since we can't follow the gradient if it's 0, the second *exploding gradients problem* since again we can't follow the gradient if it's not changing.

To solve this problem we assume that the inputs are standardized (zero mean and constant variance) and drawn from some distribution. Furthermore, we assume that $x_1, \dots, x_d$ are independent. Note that the input is the same as the hidden layer 0.

$$\mathbb{E}[x_j] = \mathbb{E}[h_j^{(0)}] = 0 \tag{5.39}$$

$$\mathbb{V}[x_j] = \mathbb{V}[h_j^{(0)}] = 1 \tag{5.40}$$

Our goal is to show that, for each activation function, we can find some distribution from which we can pick the weights of the neural network such that the standardization is preserved through each layer. In other words if the standardization of the input is preserved through all layers this means that the values nor explode neither shrink too much, *i.e.* the variance is constant. This prevents the exploding or vanishing gradient problems. We assume that all the weights are drawn from a normal distribution with zero mean and unknown variance, $W_{i,j}^{(l)} \sim \mathcal{N}(0, \sigma^2)$. Then, by induction:

$$\mathbb{E}[z_j^{(l)}] = \mathbb{E}\left[\sum_{i=1}^{d^{(l)}} h_i^{(l-1)} W_{i,j}^{(l)}\right] \tag{5.41}$$

$$= \sum_{i=1}^{d^{(l)}} \mathbb{E}\left[h_i^{(l-1)} W_{i,j}^{(l)}\right] \tag{5.42}$$

$$= \sum_{i=1}^{d^{(l)}} \mathbb{E}\left[h_i^{(l-1)}\right] \underbrace{\mathbb{E}\left[W_{i,j}^{(l)}\right]}_{=0} \tag{5.43}$$

$$= 0 \tag{5.44}$$

$$\mathbb{V}[z_j^{(l)}] = \mathbb{V}\left[\sum_{i=1}^{d^{(l)}} h_i^{(l-1)} W_{i,j}^{(l)}\right] \tag{5.45}$$

$$= \sum_{i=1}^{d^{(l)}} \mathbb{V}\left[h_i^{(l-1)} W_{i,j}^{(l)}\right] \tag{5.46}$$

$$= \sum_{i=1}^{d^{(l)}} \underbrace{\mathbb{V}\left[h_i^{(l-1)}\right]}_{=1} \underbrace{\mathbb{V}[W_{i,j}^{(l)}]}_{=\sigma^2} \tag{5.47}$$

$$= d^{(l)}\sigma^2 \tag{5.48}$$

Where we use the property that given two random variables X, Y if Y has 0 mean, then $\mathbb{V}[XY] = \mathbb{V}[X]\mathbb{V}[Y]$. Then if we pick σ^2 correctly we can mantain a variance of 1 throughout all layers.

- **TanH/Sigmoid**: Xavier Glorot normal initialization

$$W_{i,j}^{(l)} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{d^{(l-1)} + d^{(l)}}\right) \tag{5.49}$$

- **ReLU**: Kaiming He normal initialization

$$W_{i,j}^{(l)} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{d^{(l-1)}}\right) \tag{5.50}$$

5.8 Optimizers

We have seen how to update the weights of the neural network using stochastic gradient descent:

SGD Update Rule

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\boldsymbol{\theta}_t; \mathbf{x}, \mathbf{y}) \tag{5.51}$$

However SGD is not the only option.

Adaptive SGD Adaptive SGD is analog to SGD, where instead of using a fixed learning rate η_t, we change it as the number of iterations t increases. The idea is to begin with a fixed small learning rate and start decreasing it slowly after a fixed amount of iterations.

Example 5.8.1 In this example we will design an adaptive learning rate that starts as 0.1 constant and then after 1000 iterations it starts to decrease.

$$\eta_t = \min\{0.1, \frac{100}{t}\} \tag{5.52}$$

Momentum SGD Recall that neural networks are not convex, and thus it's possible that we get stuck in a local minimum or saddle point. Momentum SGD is a form of SGD where we use information about the previous step to follow the gradient even if we are temporarily stuck. The real-world analogy is that of a ball rolling down a mountain, even if there are short bumps or flat points the ball keeps rolling down because it has *momentum*.

Momentum SGD Update Rule:

$$\mathbf{v}_t \leftarrow \alpha \mathbf{v}_{t-1} - \eta_t \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\boldsymbol{\theta}_t; \mathbf{x}, \mathbf{y}) \tag{5.53}$$

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \mathbf{v}_t \tag{5.54}$$

Where $\mathbf{v}_{t-1} \in \mathbb{R}^D$ is the momentum vector that contains the retained gradient of the previous step and $\alpha \in [0, 1]$ determines how much of the momentum vector should contribute to the current update.

5.9 Overfitting

There are different countermeasures against overfitting.

1. **Early Stopping** *i.e.* don't run the optimizer until convergence, otherwise we might start to overfit to the training data. Instead every few training steps we pick some data from the validation set and check if the validation performance starts to decrease, if so we stop training. One way to do this is using a plot of the validation and training error with the number of epochs.
2. **Regularization**

5.10 Regularization

Penality Regularization As with regression methods we can regularize the model by keeping the weights small, to do so we can use a matrix norms as penality to the objective.

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \mathbb{R}^D}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) + \lambda \|\boldsymbol{\theta}\|_F \tag{5.55}$$

Definition 5.10.1 (Matrix Norm) *Let $p, q \in \mathbb{N}$, and $A \in \mathbb{R}^{m \times n}$ be a matrix then the p, q-*matrix norm *is defined as:*

$$\|A\|_{p,q} = \left(\sum_{j=1}^{n} \left(\sum_{i=1}^{m} |a_{i,j}|^p \right)^{\frac{q}{p}} \right)^{\frac{1}{q}} \tag{5.56}$$

Using the matrix norm we can define the *Frobenius* norm which is the most popular norm used to regularize the weights $\boldsymbol{\theta}$, it's the analogous of the $L2$-norm but with matrices.

Definition 5.10.2 (Frobenius Norm) *If $p = 2$, $q = 2$, and $A \in \mathbb{R}^{m \times n}$ the matrix norm is called* Frobenius *norm and is defined as:*

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{i,j}|^2} = \sqrt{\text{trace}(A^H A)} \tag{5.57}$$

Similarly if we set $p = 1$, $q = 1$ we would get the analogous to the $L1$-norm. As always the value of λ, which weights the importance of regularization, is found using cross-validation.

Dropout Overfitting happens because the training data is a noisy approximation of the real underlying distribution, thus some neurons in the hidden layers of our neural network will start to recognize noise instead of general characteristics of the data. To solve this problem theoretically, we could train many neural networks to get different values of $\boldsymbol{\theta}$ and then average the learned parameters. The problem with this approach is that if there are a lot of parameters training a neural network many times takes a prohibitive amount of time. To solve this problem, instead of training many networks, we only train a single one but apply a neat trick called *dropout*. Dropout is a form of regularization that is not applied to the objective function but is applied directly inside of the

network, more precisely on some hidden layer. Recall how to evaluate the current hidden layer given the previous one.

$$\mathbf{h}^{(l)} := \varphi^{(l)}\left(\mathbf{h}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}\right) \tag{5.58}$$

$$\tag{5.59}$$

The idea is that during training we set some amount of neurons of the hidden layer to 0 (*i.e.* $h_j^{(l-1)} = 0$ for some j) with probability p, this simulates the training of only a single sub-network. This has many effects, the first is that the network is forced to learn a representation that is more sparse (*i.e.* to not use the entire hidden layer all the times), the second is that if some neurons previously were overfitting to noise now that they are deactivated they are forced to store only important features. Furthermore, activating only a single set of hidden units has a similar effect to training multiple networks and averaging the values of the weights. To avoid overfitting using dropout we change the hidden layer evaluation slightly as follows:

$$\mathbf{d}^{(l-1)} \sim \text{Bernoulli}(1-p) \tag{5.60}$$

$$\mathbf{h}^{(l)} := \varphi^{(l)}\left((\mathbf{h}^{(l-1)} \odot \mathbf{d}^{(l-1)})\mathbf{W}^{(l)} + \mathbf{b}^{(l)}\right) \tag{5.61}$$

Where $\mathbf{d}$ is the dropout vector that randomly sets some values of the hidden layer to 0. Note that each training step will select a different set of hidden neurons with probability p and that during validation dropout is disabled, *i.e.* $p = 0$. Depending on the architecture we can apply dropout to only a few layers or to all of them.

Batch Normalization Recall that using stochastic gradient descent with a simple training sample might have a large variance and thus lead to a slow convergence. We have seen that training with mini-batches of data ($\mathbf{X}_{1\dots B} = [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_B}]^T, \mathbf{Y}_{1\dots B} = [\mathbf{y}_{i_1}, \dots, \mathbf{y}_{i_B}]^T$) ($B$ samples in the mini-batch) helps to solve this problem. Note that by the way neural networks are built, if instead of a single vector we pass a matrix $\mathbf{X}_{1\dots B}$ with features in the columns and different elements of the batch in the rows as input, we will get as output the entire batch prediction, *i.e.* $f(\mathbf{X}_{1\dots B}, \mathbf{Y}_{1\dots B}; \boldsymbol{\theta}) = \hat{\mathbf{Y}}_{1\dots B}$, without having to loop through each data sample. The Batch-SGD step is then:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \frac{1}{B} \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\boldsymbol{\theta}_t; \mathbf{X}_{1\dots B}, \mathbf{Y}_{1\dots B}) \tag{5.62}$$

Usually is a good practice to standardize the input data in each batch ($\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_B}$) to avoid the vanishing and exploding gradient problem. The idea of *batch normalization* is that we can not only standardize the input data (and thus the hidden layer 0), but also each intermediate hidden layer. Similarly to the dropout regularization technique, batch normalization is a regularization technique applied to each layer of the network. Batch normalization has many advantages, it helps to keep the weights of the neural network small, it enables faster training with higher learning rates, and overall improves the stability of the network. To apply batch normalization to layer l, we will normalize the output of the hidden layer using the data from the entire batch. If we feed a batch as input, the hidden layer l, $\mathbf{h}^{(l)} = \phi^{(l)}(\mathbf{z}^{(l)})$, will be of dimension

$\mathbf{h}^{(l)}, \mathbf{z}^{(l)} \in \mathbb{R}^{d^{(l)} \times B}$. Then we apply batch normalization on $\mathbf{z}^{(l)}$:

$$\boldsymbol{\mu}_B^{(l)} \coloneqq \frac{1}{B} \sum_{j=1}^{B} \mathbf{z}_{:,j}^{(l)} \qquad \textit{Batch mean} \qquad (5.63)$$

$$\boldsymbol{\sigma}_B^{2\,(l)} \coloneqq \frac{1}{B} \sum_{j=1}^{B} (\mathbf{z}_{:,j}^{(l)} - \mu_B^{(l)}) \qquad \textit{Batch standard deviation} \qquad (5.64)$$

$$\tilde{\mathbf{z}}^{(l)} \coloneqq \frac{\mathbf{z}^{(l)} - \boldsymbol{\mu}_B^{(l)}}{\sqrt{\boldsymbol{\sigma}_B^{2\,(l)} + \boldsymbol{\varepsilon}}} \qquad \textit{Standardization} \qquad (5.65)$$

$$\mathrm{BN}_{\boldsymbol{\gamma}^{(l)}, \boldsymbol{\beta}^{(l)}}(\mathbf{z}^{(l)}) \coloneqq \boldsymbol{\gamma}^{(l)} \odot \tilde{\mathbf{z}}^{(l)} + \boldsymbol{\beta}^{(l)} \qquad \textit{Batch normalization} \qquad (5.66)$$

Where $\boldsymbol{\varepsilon} \in \mathbb{R}^{1 \times d^{(l)}}$ is a small number added to avoid division by 0, and $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{1 \times d^{(l)}}$ are two variables that will be trained along with $\boldsymbol{\theta}$, *i.e.* $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} \cup \{\boldsymbol{\gamma}, \boldsymbol{\beta}\}$ to automatically denormalize our data. The idea is that by optimizing $\boldsymbol{\gamma}, \boldsymbol{\beta}$ the network won't have to destabilize itself by increasing disproportionately its weights, but solely these two parameters. Finally we can change the evaluation of layer l to:

$$\mathbf{h}^{(l)} = \varphi^{(l)}(\mathrm{BN}_{\boldsymbol{\gamma}^{(l)}, \boldsymbol{\beta}^{(l)}}(\mathbf{z}^{(l)})) \qquad (5.67)$$

$$= \varphi^{(l)}(\mathrm{BN}_{\boldsymbol{\gamma}^{(l)}, \boldsymbol{\beta}^{(l)}}(\mathbf{h}^{(l-1)}\mathbf{W}^{(l)})) \qquad (5.68)$$

Note that when using batch normalization the bias $\mathbf{b}^{(l)}$ is not necessary since it's already implemented by $\boldsymbol{\beta}$.

5.11 Convolutional Neural Networks (CNNs)

Convolutional neural networks are very similar to standard neural networks. CNNs are usually used for image recognition, if we used a standard neural network to recognize images there would be some problems. The first is that would have to input a vector of the same size as the number of pixels, even small images *e.g.* 200x200x3 (200x200 pixels with 3 color channels) would use 120000 different inputs. Secondly, images are a special type of data, most neighboring pixels are correlated with each other and thus contain redundant information, usually the greatest amount of information is found when there are big differences between neighboring pixels. Convolutional neural networks try to solve those problems by using trainable *convolution* filters to compress the representation of the grid of pixels and keeping only important features. Note that CNNs work well for all types of applications that have spatially correlated data, *i.e. d*-dimensional grids of values where neighboring data-points are correlated, hence not only for images. The general idea behind CNNs is that we start with a grid of values, we apply one or multiple filters on it, and then use this new representation as input to a standard feed forward neural network.

Transformation Invariance When dealing with spatially correlated data (think of an image, or 2 dimensional grid, of a digit *e.g.* '4'), we usually want the information to be recognized even if different transformations are applied.

- *Shift invariance*, if the important information is shifted (*e.g.* if the digit '4' is not centered in the image but on the top left).
- *Rotation invariance*, if the important information is rotated (*e.g.* digit '4' rotated by a few degrees).
- *Scale invariance*, if the important information is concentrated in a small area or on the entire grid (*e.g.* digit '4' is written in a small font, or big font).

Thus we want our model to be resistant to invariance. There are different ways to solve this problem, one might be to use a standard neural network and train it not only with the standard data, but also with the transformed data (shifted, rotated, scaled). This process is called data *augmentation*. Other methods use special types of regularization, but the state of the art is usually only achieved by using CNNs.

Convolution Operation The convolution operation is used to filter important correlation between neighboring cells and also to transform the input. We have seen that convolutions applied on grid of values. The idea is that we have one or more convolutional filters, which are just matrices of numbers, that will be sliding over the grid and extract the important features. To precisely describe a convolution we will have to set the following parameters:

- *Batch size*: B, similarly to neural networks we can apply to convolution to one batch or many batches at the same time (*e.g.* if we apply it to a single image at the time $B = 1$, if many images at the time $B > 1$).
- *Filters*: F, during a convolution operation a filter is applied on the image, we can apply a single filter at the time *i.e.* $F = 1$, or many filters at the time *i.e.* $F > 1$.
- *Dimension*: $D_w,\ D_h,\ D_d$, the dimension of the input (*e.g.* a 2D image of 20×30 pixels will have $D_w = 20,\ D_h = 30$, or a 1D audio signal with 100 discrete timesteps will have $D_w = 100$).
- *Channels*: C, the number of channels of the signal (*e.g.* a 2D color image in RGB will have $C = 3$ color channels, and a 1D audio signal will have $C = 2$ different sound channels). Note that sometimes the number of channel is added as an additional dimension, however with standard application it's good practice to keep it separate (*e.g.* a video will have $D_w, D_h, D_d = 3$ for width, height and time, and $C = 3$ for the color channels).
- *Kernel Size*: $K_w,\ K_h,\ K_d$, the dimension of the filter (*e.g.* $K_w = 3,\ K_h = 3$).
- *Padding*: $P_w,\ P_h,\ P_d$, sometimes is convenient to pad the input volume with zeros so that we can control the output dimension (*e.g.* in 2D with $P_w,\ P_h$ we go to $D_w \leftarrow D_w + 2P_w,\ D_h \leftarrow D_h + 2P_h$. Times 2 since we pad top/bottom respectively left/right).
- *Stride*: $S_w,\ S_h,\ S_d$, by how much we move the filter window, usually $S = 1$. The higher the stride the a smaller output representation we will get. (*e.g.* a stride of $S = 2$ will slide the filter at intervals of 2 pixels on the image).

Then each filter will be applied to all batches of data and we will get as output a matrix of size $B \times F \times \tilde{D}_w \times \tilde{D}_h \times \tilde{D}_d$. Where $\tilde{D}_\star$ depends on

the input dimension, padding and stride. Note that matrices with more than 2 dimensions are usually referred to as *tensors*.

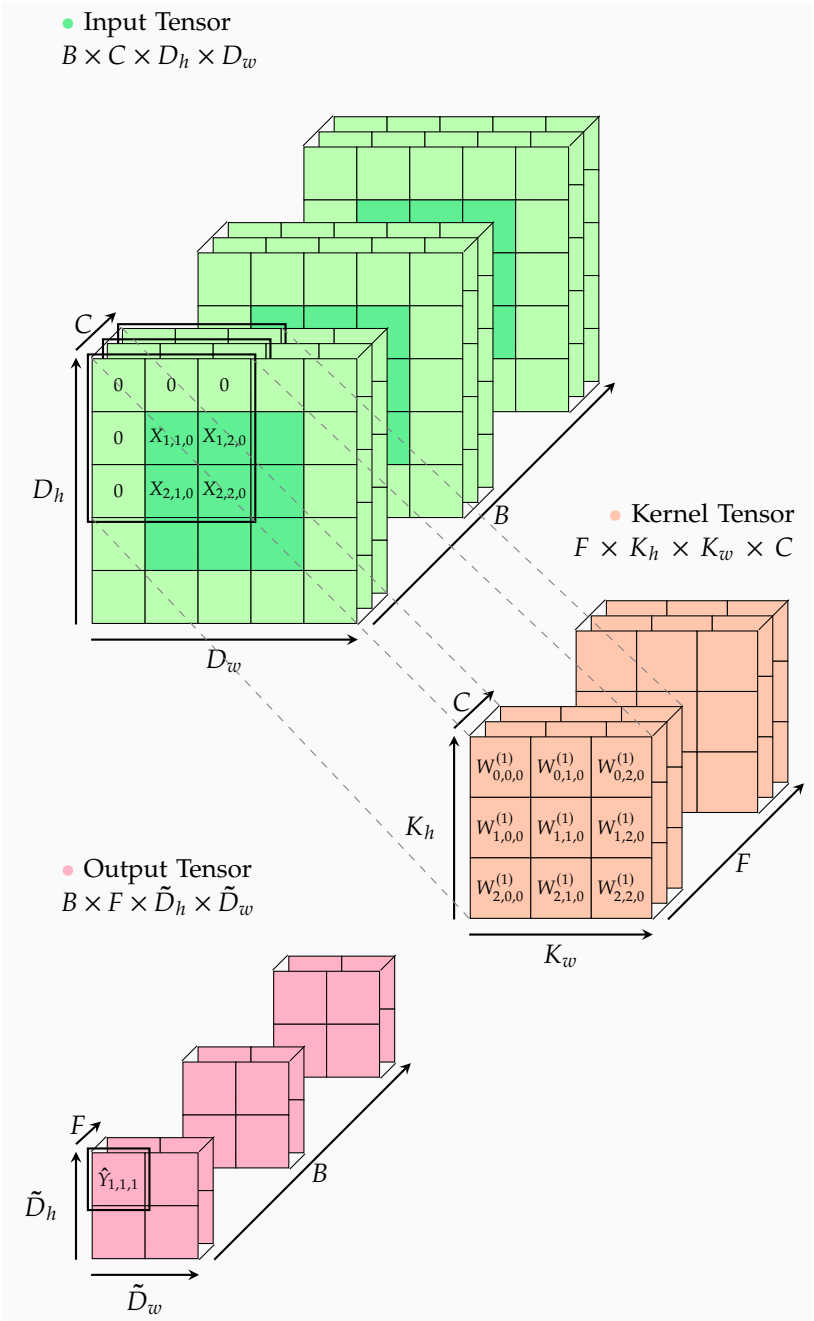
$$\tilde{D}_\star = \left\lfloor \frac{D_\star + 2P_\star - K_\star}{S_\star} + 1 \right\rfloor \tag{5.69}$$

Definition 5.11.1 (Convolution) *Given a 2D input matrix* $\mathbf{X} \in \mathbb{R}^{D_w \times D_h}$ *with* C *channels and a kernel filter matrix*[a] $\mathbf{W}^{(f)} \in \mathbb{R}^{K_w \times K_h}$ *with* $f \in \{1, \ldots, F\}$, *the* convolution $\hat{\mathbf{Y}}^{(f)} := (\mathbf{X} * \mathbf{W}^{(f)})_{i,j}$ *between* $\mathbf{X}$ *and* $\mathbf{W}^{(f)}$ *at* i, j *is defined as:*

$$\hat{Y}_{i,j,f} := \sum_{x=0}^{K_w-1} \sum_{y=0}^{K_h-1} \sum_{c=0}^{C-1} X_{i+x,j+y,c} W_{x,y,c}^{(f)} \tag{5.70}$$

[a] We use the notation $\mathbf{W}$ to make clear that this a trainable weight matrix.

Example 5.11.1 (2D-Convolution) Let the following convolution be applied on a 2D color image: $B = 3$ (3 images at the time), $F = 2$ (2 filters applied on each image), $D_w, D_h = 5$ (5×5 image size), $C = 3$ (color channels), $K_x, K_y = 3$ (3×3 kernel size) $P_x, P_y = 1$ (1 zero pad), $S_x, S_y = 2$ (2 stride) Then to compute the convolution $\hat{\mathbf{Y}}$ we slide all filters over all images in the batch, and at each step multiply all overlapping cells and sum them together. The following illustration shows computation of the convolution step for $\hat{Y}_{1,1}^{(1)}$.



We have 3 batches and 2 filters, each batch is applied to all filters thus we have a total of 6 convoluted images ($B \cdot F$). For example the index $1, 1$ of the first convoluted images is evaluated as:

$$\hat{Y}_{1,1,1} = X_{1,1,0}W^{(1)}_{1,1,0} + X_{1,2,0}W^{(1)}_{1,2,0} + X_{2,1,0}W^{(1)}_{2,1,0} + X_{2,2,0}W^{(1)}_{2,2,0} \quad (5.71)$$

$$+ X_{1,1,1}W^{(1)}_{1,1,1} + X_{1,2,1}W^{(1)}_{1,2,1} + X_{2,1,1}W^{(1)}_{2,1,1} + X_{2,2,1}W^{(1)}_{2,2,1} \quad (5.72)$$

$$+ X_{1,1,2}W^{(1)}_{1,1,2} + X_{1,2,2}W^{(1)}_{1,2,2} + X_{2,1,2}W^{(1)}_{2,1,2} + X_{2,2,2}W^{(1)}_{2,2,2} \quad (5.73)$$

Note that for this cell we sum only 4 values for each channel since the input tensor is zero padded.

We start with images of size $C \times D_h \times D_w$ and after the convolution each of those images will have size $F \times \tilde{D}_h \times \tilde{D}_w$, thus in a sense the

number of filters F is the same as the number of output channels *i.e.* $\tilde{C} := F$. Going back to the convolution operation, we could either put the current representation into a vector [2] $\mathbf{x} \in \mathbb{R}^{\tilde{D}_h \cdot \tilde{D}_w \cdot \tilde{C}}$ and feed it as input to a standard a neural network, or apply more layers of operations using the output channel $\tilde{C}$ as the new input channel C.

2: The operation of putting a tensor into a vector is called *flattening*.

Pooling Operation Convolutional layers work well to extract important correlation, but are not designed to compress the representation. If we want to reduce the number of parameters that we feed to the neural network by a reasonable amount usually we apply an additional operation to the output of the convolution called *pooling*. Pooling subsamples the given representation, usually it's an operation that splits the grid into subgrids, returns either the average or the maximum of each subgrid. Similarly to convolutions, we can still change the padding, kernel size, or stride even if usually $S_w = K_w$, $S_h = K_h$ but this time the number of channels stays fixed.

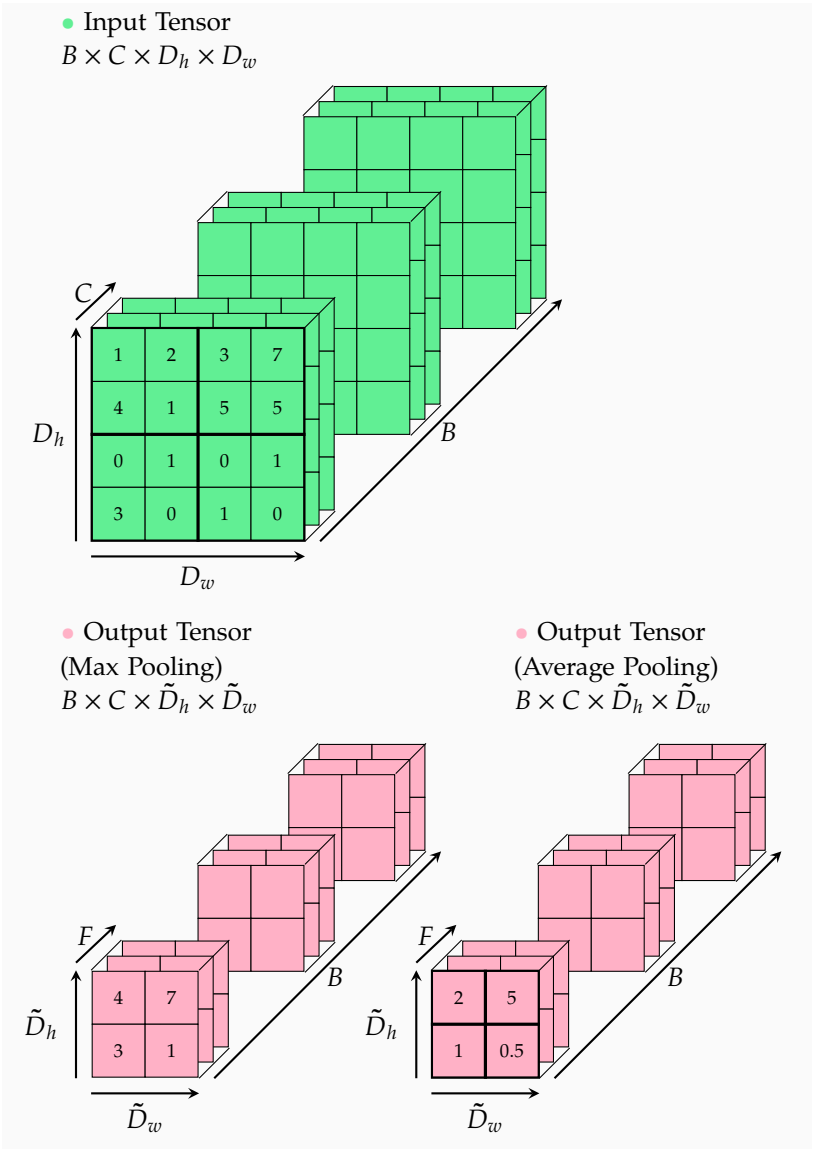
Definition 5.11.2 (Max/Average Pooling) *Max Pooling:*

$$\hat{Y}_{i,j,c} := \max_{x=0,\dots,K_w-1} \max_{y=0,\dots,K_h-1} X_{i+x,j+y,c} \tag{5.74}$$

Avg. Pooling:

$$\hat{Y}_{i,j,c} := \frac{1}{K_w K_h} \sum_{x=0}^{K_w-1} \sum_{y=0}^{K_h-1} X_{i+x,j+y,c} \tag{5.75}$$

Example 5.11.2 (Pooling) Let the following pooling operation be applied on a 2D color image: $B = 3$ (3 images at the time), $D_w, D_h = 4$ (4×4 image size), $C = 3$ (color channels), $K_x, K_y = 2$ (2×2 kernel size) $P_x, P_y = 0$ (0 zero pad), $S_x, S_y = 2$ (2 stride) Then to compute the pooling we take subgrids of 2 at the time and average or take the maximum of each.



Operation Stacking In a real-world scenario, we might apply many layers of convolution and pooling until the last layer returns a tensor of size $\tilde{C} \times 1 \times 1$ with a high dimensional $\tilde{C}$ that will then flattened into a vector and fed to a standard neural network. If we then visualize what each filter matrix $\mathbf{W}^{(f)}$ is doing layer by layer we will see that at, as more layers are applied, more and more complex features will be recognized. If, for example, we are doing face recognition we will start with vertical, horizontal lines and edges, then we recognize curves, then circles, then heads eyes and noses, and finally faces. Thus if by using standard feed forward neural network each neuron was recognizing a global pattern, with convolutional neural networks we can use the convolutional layers to recognize useful *local* patterns.

5.12 Other

Weight-Space Symmetry All the weights of a neural network are symmetric, thus there are exponentially many possible local minima that

might be in fact global minima.

Kernels vs NNs The tradeoffs are:

Kernels

- + Convex optimization *i.e.* no local minima.
- + Robust against noise.
- +/- Models grow with the size of data.
- - Don't allow multiple layers.

NNs

- + Flexible non-linear models with fixed parametrization.
- + Mutiple layers discover representation at different levels of abstraction
- - Many hyperparameters that have to be tuned.
- - Suffer if data is noisy (*i.e.* regularization is often essential).

How to choose hyperparamters?

Probabilistic Approach to Supervised Learning

6

So far we have discussed how we can fit prediction models (both linear and non-linear) for regression and classifications problems without any statistical interpretation. Often we would like to statistically model the data in order to quantify uncertainty and being able to express prior knowledge or assumptions about the data. In this chapter, we will see how many of the approaches we have already discussed can be interpreted as fitting probabilistic models. This view will allow us to derive new methods.

So far, given a training data, we wanted to identify an hypothesis (*e.g.* linear model, kernelized models, neural networks, ...) in order to minimize the prediction error. Our fundamental assumption was that the data set is generated independently and identically distributed from a probability distribution $P(\mathbf{x}, y)$. Unfortunately, this probability distribution is unknown, otherwise supervised learning would simply mean finding the hypothesis which minimizes (in terms of a loss function l) the following expression:

$$R(h) = \int P(\mathbf{x}, y) l(y, h(\mathbf{x})) d\mathbf{x} dy = \mathbb{E}_{\mathbf{x},y} \left[l(y; h(\mathbf{x})) \right]$$

Now we want to answer the following question: *what is the upper bound, with the best possible hypothesis, that one can achieve?* The following lemma gives the solution for the case of square loss, but the same idea generalizes also to other loss functions.

Lemma 6.0.1 *If one knows $P(\mathbf{x}, y)$ and assumin that the data are generated iid from such a distribution, the best possible hypothesis predicts*

$$h^*(x) = \mathbb{E}\left[Y | X = x\right]$$

This, in practice unattainable, hypothesis is called ***Bayes'optimal predictor*** *for the square loss.*

Proof. We have:

$$\begin{aligned}
\min_h R(h) &= \min_h \mathbb{E}_{\mathbf{X}, Y \sim P} \left[(Y - h(\mathbf{X})^2 \right] \\
&= \min_h \mathbb{E}_{\mathbf{X}} \left[\mathbb{E}_Y \left[(Y - h(\mathbf{x}))^2 | \mathbf{X} = \mathbf{x} \right] \right] \\
&= \mathbb{E}_{\mathbf{X}} \left[\min_{h(\mathbf{x})} \mathbb{E}_Y \left[(Y - h(\mathbf{x}))^2 | \mathbf{X} = \mathbf{x} \right] \right]
\end{aligned}$$

Hence the best possible hypothesis is the one which finds the optimal

prediction $y^*(x)$ defined as:

$$y^*(\mathbf{x}) \in \arg\min_{\hat{y}} \mathbb{E}_Y \left[(\hat{y} - Y)^2 | \mathbf{X} = \mathbf{x}\right]$$

This means that we formally have to minimize the following expression:

$$l(\hat{y}) := \int (\hat{y} - y)^2 \cdot p(y|\mathbf{x}) dy$$

Which we can do by taking the derivative with respect to $\hat{y}$ and set it to zero. One get that the necessary condition is given by:

$$\int \hat{y} \cdot p(y|\mathbf{x}) dy = \int y \cdot p(y|\mathbf{x}) dy$$

Where the first part is equal to $\hat{y}$ and the second is $\mathbb{E}[Y|\mathbf{X} = \mathbf{x}]$ □

We will now study least square estimations under a statistical perspective. Before we dive into the analysis, we need to define some concepts.

Definition 6.0.1 (Parametric estimation) *A parametric estimation is of $\mathbb{P}[Y|X]$ is a parametric formula of the form*

$$\hat{\mathbb{P}}[Y|\mathbf{X}, \Theta]$$

Definition 6.0.2 (Maximum conditional likelihood estimation) *We want to estimate the optimal value of Θ, i.e.*

$$\begin{aligned}
\Theta^* &= \arg\max_{\Theta} \hat{\mathbb{P}}\left[y_1, \ldots, y_n | \mathbf{x}_1, \ldots, \mathbf{x}_n, \Theta\right] \\
&= \arg\max_{\Theta} \prod_{i=1}^{n} \hat{\mathbb{P}}\left[y_i | x_i, \Theta\right] \\
&= \arg\max_{\Theta} \log \prod_{i=1}^{n} \hat{\mathbb{P}}\left[y_i | x_i, \Theta\right] \\
&= \arg\max_{\Theta} \sum_{i=1}^{n} \log \hat{\mathbb{P}}\left[y_i | \mathbf{x}_i, \Theta\right] \\
&= \arg\min_{\Theta} \sum_{i=1}^{n} \log \hat{\mathbb{P}}\left[y_i | \mathbf{x}_i, \Theta\right]
\end{aligned}$$

Lemma 6.0.2 *Under the conditional linear Gaussian assumption, maximizing the likelihood is equivalent to least square optimization. That is, if we assume*

$$y_i \sim \mathcal{N}(\mathbf{w}^T \mathbf{x}_i, \sigma^2)$$

then we get

$$\arg\max_{\mathbf{W}} \mathbb{P}\left[y_1, \ldots, y_n | \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{w}\right] = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} (y_i - \mathbf{w}\mathbf{x}_i)^2$$

Proof. We have:

$$\begin{aligned}
&\arg\max_{\mathbf{W}} \mathbb{P}\left[y_1, \ldots, y_n | \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{w}\right] \\
&= \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} \log \hat{\mathbb{P}}\left[y_i | \mathbf{x_i}, \mathbf{w}\right] \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \left(\frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y_i - \mathbf{w}\mathbf{x}_i)^2\right) \\
&= \arg\min_{\mathbf{w}} \frac{n}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mathbf{w}\mathbf{x}_i)^2 \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n}(y_i - \mathbf{w}\mathbf{x}_i)^2
\end{aligned}$$

□

Hence we have that the maximum likelihood estimator is given by the least square solution, assuming that the noise is *i.i.d.* Gaussian with constant variance. This is useful since the maximum likelihood estimator satisfies several nice statistical properties such as consistency (parameter estimate converges to true parameters in probability), asymptotic efficiency (smallest variance among all well-behaved estimators for large n) and asymptotic normality. Keep in mind that those properties are asymptotic (*i.e.* they hold for $n \to \infty$). For finite n is crucial to avoid overfitting!

6.1 Bias Variance Tradeoff

Definition 6.1.1 (Bias) *Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. Models with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data. Bias can also be considered as the excess risk of best model compared to minimal achievable risk knowing* $\mathbb{P}[\mathbf{X}, \mathbf{Y}]$. *Formally:*

$$\mathbb{E}_{\mathbf{X}}\left[\mathbb{E}_D \hat{h}_D(\mathbf{X}) - h^*(\mathbf{X})\right]$$

Definition 6.1.2 (Variance) *Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. Model with high variance pays a lot of attention to training data and does not generalize on the data which it has not seen before. As a result, such models perform very well on training data but has high error rates on test data. This is the risk incurred due to estimating model from limited data. Formally:*

$$\mathbb{E}_{\mathbf{X}} Var_D\left[\hat{h}_D(\mathbf{X})\right]^2 = \mathbb{E}_{\mathbf{X}}\mathbb{E}_D\left[\hat{h}_D(\mathbf{X}) - \mathbb{E}_{D'}\hat{h}_{D'}(\mathbf{X})\right]^2$$

Definition 6.1.3 (Noise) *Irreducible error, formally defined as*

$$\mathbb{E}_{\mathbf{X},\mathrm{Y}}\left[(\mathrm{Y} - h^*(\mathbf{X}))^2\right]$$

Lemma 6.1.1 (Bias variance tradeoff) *The expected prediction error is given by*

$$Bias^2 + Variance + Noise$$

Proof. We have:

$$\begin{aligned}\mathbb{E}_{\mathbf{X}}\left[\mathbb{E}_D\hat{h}_D(\mathbf{X}) - h^*(\mathbf{X})\right]^2 &= \mathbb{E}_{\mathbf{X}}Var_D\left[\hat{h}_D(\mathbf{X})\right]^2 \\ &+ \mathbb{E}_{\mathbf{X}}\mathbb{E}_D\left[\hat{h}_D(\mathbf{X}) - \mathbb{E}_{D'}\hat{h}_{D'}(\mathbf{X})\right]^2 \\ &+ \mathbb{E}_{\mathbf{X},\mathrm{Y}}\left[(\mathrm{Y} - h^*(\mathbf{X}))^2\right]\end{aligned}$$

□

Ideally, we wish to find an estimator that minimizes both bias and variance. However one should keep this idea in mind: the bias is a decreasing function in terms of model complexity (*i.e.* with very complex models we can achieve a very small bias), while the variance is an increasing function in terms of model complexity (*i.e.* with a very complex model the variance increases). Hence we need a model with a good balance between bias and variance in order to minimize the prediction error. We have that the maximum likelihood estimator (*i.e.* least squares) for linear regression is unbiased. In facts, by choosing a proper polynomial degree, we can fit all possible data sets. Moreover, as stated in the Gauss-Markov theorem, this is the minimum variance estimator among all unbiased ones. However, we have already seen that the least squares solution can overfit. Thus we trade a little bit of bias for a potentially dramatic reduction of variance. We have discussed regularization as a solution in this sense. But how do this kind of tricks fit into the probabilistic view of the situation?

The basic idea of regularization is penalizing large weights because we believe that those are an indicator of overfitting. Hence we are implicitly introducing assumptions about the weights, we *assume* that weights will probably not be too large. From the statistical perspective, we can achieve the same result by introducing prior assumptions about the probability distribution.

Lemma 6.1.2 *Ridge regression can be understood as finding the Maximum a Postetiori (MAP) parameter estimate for a linear regression problem assuming that the noise* $\mathbb{P}\left[y|\mathbf{w},\mathbf{x}\right]$ *is i.i.d. Gaussian and the prior* $\mathbb{P}\left[\mathbf{w}\right]$ *on the model parameters* $\mathbf{w}$ *is Gaussian.*

Proof. First we need to compute the posterior distribution of $\mathbf{w}$ using the Bayes'rule

$$\begin{aligned}\mathbb{P}\left[\mathbf{w}|\mathbf{x}_{1:n}, y_{1:n}\right] &= \frac{\mathbb{P}\left[\mathbf{w}|\mathbf{x}_{1:n}\right]\mathbb{P}\left[y_{1:n}|\mathbf{w},\mathbf{x}_{1:n}\right]}{\mathbb{P}\left[y_{1:n}|x_{1:n}\right]} \\ &= \frac{\mathbb{P}\left[\mathbf{w}\right]\mathbb{P}\left[y_{1:n}|\mathbf{w},\mathbf{x}_{1:n}\right]}{\mathbb{P}\left[y_{1:n}|x_{1:n}\right]}\end{aligned}$$

We have to find the weights $\mathbf{w}$ that maximize the expression, with the assumption that they are normally distributed with mean zero and $\sigma^2 = \beta^2$. We get

$$\arg\max_{\mathbf{w}} \mathbb{P}\left[\mathbf{w}|\mathbf{x}_{1:n}, y_{1:n}\right] = \arg\min_{\mathbf{w}} -\log \mathbb{P}\left[\mathbf{w}\right] - \log \mathbb{P}\left[y_{1:n}|\mathbf{w}, \mathbf{x}_{1:n}\right] + \log \mathbb{P}\left[y_{1:n}|\mathbf{x}_{1:n}\right]$$

where the second term is equal to $\arg\min_{\mathbf{w}} \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - \mathbf{w}\mathbf{x}_i)^2$ as shown in the proof of Lemma 6.0.2 and the third term does not depend on $\mathbf{w}$. Hence we have to work on the first term only

$$\begin{aligned}
-\log \mathbb{P}\left[\mathbf{w}\right] &= -\log \prod_{i=1}^{d} \mathbb{P}\left[w_i\right] \\
&= -\sum_{i=1}^{d} \log\left(\frac{1}{\sqrt{2\pi\beta^2}} exp(-\frac{w_i}{2\beta^2})\right) \\
&= \frac{d}{2}\log 2\pi\beta^2 + \frac{1}{2\beta^2}\sum_{i=1}^{d} w_i^2 \\
&= \frac{1}{2\beta^2}||\mathbf{w}||_2^2 + \mathcal{O}(\mathbf{1})
\end{aligned}$$

Hence our problem reduces to

$$\begin{aligned}
&\arg\min_{\mathbf{w}} \frac{1}{2\beta^2}||vecw||_2^2 + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i)^2 \\
&= \arg\min_{\mathbf{w}} \frac{\sigma^2}{\beta^2}||\mathbf{w}||_2^2 + \sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i)^2
\end{aligned}$$

which is ridge regression with parameter $\lambda := \frac{\sigma^2}{\beta^2}$ □

by changing our assumption regarding the distribution of the weights we get different regularizers, *e.g.* the Laplace distribution is the prior of Lasso regression.

6.2 Logistic Regression

So far we have discussed a probabilistic approach to regression. What can we say about classification? In classification the risk is, considering accuracy as metric, given by:

$$R(h) = \mathbb{E}_{\mathbf{X},Y}\left[[Y \neq h(\mathbf{X})]\right]$$

If we unrealistically suppose that we knew $P(\mathbf{X}, Y)$, the h that minimizes the risk would be given by the one that outputs the most probable class, i.e.

$$h^*(x) = \arg\max_{y} \mathbb{P}\left[Y = y|\mathbf{X} = \mathbf{x}\right]$$

A new model for classification is logistic regression, where we estimate the probability that a given sample belongs to a certain class. We want that a realisation that is positive and far away from the border will have

a very high probability, a realisation that is negative and far away from the border will have a probability close to zero, and cases close to the border will have a probability of around 0.5.

Definition 6.2.1 (Logistic regression) *Logistic regression is a classification method that estimates the probability that an input is in the positive class. Formally*

$$\mathbb{P}\left[Y = y|\mathbf{x}\right] = \sigma'(\mathbf{w}^T\mathbf{x}) = \frac{1}{1 + exp(y\mathbf{w}^T\mathbf{x})}$$

We replace the assumption of Gaussian noise that we used for regression with i.i.d. Bernoulli noise, i.e.

$$\mathbb{P}\left[y|\mathbf{w}, \mathbf{x}\right] = Ber(y; \sigma'(\mathbf{w}^T\mathbf{x}))$$

Lemma 6.2.1 *The maximum likelihood estimator for logistic regression is given by*

$$\hat{R}(\mathbf{w}) = \sum_{i=1}^{n} \log(1 + exp(-y_i\mathbf{w}^T\mathbf{x}_i))$$

Proof. We have

$$\begin{aligned}
\hat{\mathbf{w}} \in \arg\max_{\mathbf{w}} \mathbb{P}\left[D|\mathbf{w}\right] &= \arg\max_{\mathbf{w}} \prod_{i=1}^{n} \mathbb{P}\left[y_i|\mathbf{x}_i, \mathbf{w}\right] \\
&= \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} \log \mathbb{P}\left[y_i|\mathbf{x}_i, \mathbf{w}\right] \\
&= \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} -\log \frac{1}{1 + exp(-y_i\mathbf{w}^T\mathbf{x}_i)} \\
&= \arg\min_{w} \sum_{i=1}^{n} \log(1 + exp(y_i\mathbf{w}^T\mathbf{x}_i))
\end{aligned}$$

□

A good property of the logistic loss is convexity, hence we can use stochastic gradient descent in order to find (an arbitrarily good approximation of) optimal weights.

Lemma 6.2.2 *The gradient of the logistic loss is given by*

$$\begin{aligned}
\nabla_{\mathbf{W}} \log(1 + exp(-y\mathbf{w}^T\mathbf{x})) &= \frac{1}{1 + exp(-y\mathbf{w}^T\mathbf{x}} \cdot exp(-y\mathbf{w}^T\mathbf{x}) \cdot (-y\mathbf{x}) \\
&= \frac{exp(-y\mathbf{w}^T\mathbf{x}}{1 + \exp(-y\mathbf{w}^T\mathbf{x}} \cdot (-y\mathbf{x}) \\
&= \frac{1}{1 + exp(y\mathbf{w}^T\mathbf{x}} \cdot (-y\mathbf{w}) \\
&= -y\mathbf{x}\mathbb{P}\left[Y \neq -y|\mathbf{w}, \mathbf{x}\right]
\end{aligned}$$

Of course, in order to avoid overfitting, one can use a regularizer (either with L2 or L1 norm) and modify the above gradient accordingly. Once

the model is trained one can compute the following in order to do the classification:

$$\begin{aligned}\arg\max_{\hat{y}} \mathbb{P}\left[\hat{y}|\mathbf{x},\mathbf{w}\right] &= \arg\max_{\hat{y}} \frac{1}{1+exp(-\hat{y}\mathbf{w}^T\mathbf{x})} \\ &= \arg\min_{\hat{y}} exp(-\hat{y}\mathbf{w}^T\mathbf{x}) \\ &= \arg\min_{\hat{y}} -\hat{y}\mathbf{w}^T\mathbf{x} \\ &= \arg\max_{\hat{y}} \hat{y}\mathbf{w}^T\mathbf{x}\end{aligned}$$

Logistic regression has some important variants that are worth mentioning:

- Kernelized logistic regression: find optimal weights with

$$\hat{\alpha} = \arg\min_{\alpha} \sum_{i=1}^{n} \log(1+exp(-y_i\alpha\mathbf{K}_i)) + \lambda\alpha^T\mathbf{K}\alpha$$

  where $\mathbf{K}_i$ is the $i-$th column of the kernel matrix.
- Multi-class logistic regression: maintain one weight vector per class and estimate the probability of each class as

$$\mathbb{P}\left[Y=i|\mathbf{x},\mathbf{w}_1,\ldots,\mathbf{w}_c\right] = \frac{exp(\mathbf{w}_i\mathbf{x}}{\sum_{j=1}^{c} exp(\mathbf{w}_j\mathbf{x})}$$

  which corresponds to the loss function

$$l(y;\mathbf{x},\mathbf{w}_1,\ldots,\mathbf{w}_c) = -\log\mathbb{P}\left[Y=y|\mathbf{x},\mathbf{w}_1,\ldots,\mathbf{w}_c\right]$$

Logistic regression is a classification method. So far we have discussed other classification methods such as SVMs. A drawback of logistic regression compared to SVMs is that obtained solutions are often dense, but with logistic regression is very easy to obtain class probabilities.

6.3 Bayesian Decision Theory

So far we have seen how we can interpret supervised learning as fitting probabilistic models of the data. Now we will discuss a framework that allows one to pick the best decision under uncertainty with the estimated models.

Definition 6.3.1 (Bayesian Decision Theory) *Given a conditional distribution over labels $\mathbb{P}\left[y|\mathbf{x}\right]$, a set of actions $\mathcal{A}$ and a cost function $\mathcal{C}: Y\times\mathcal{A}\rightarrow\mathbb{R}$, Bayesian decision theory recommends to pick the action that minimizes the expected cost*

$$a^* = \arg\min_{a\in\mathcal{A}} \mathbb{E}_y\left[\mathcal{C}(y,a)|\mathbf{x}\right]$$

In general, if we had access to the true distribution $\mathbb{P}\left[y|\mathbf{x}\right]$, this implements the Bayesian optimal decision. In practice, this probability can only be estimated, *e.g.* with logistic regression.

Example 6.3.1 Suppose one has estimated a logistic regression model for spam filtering and has obtained a probability p for a given message to be spam. Further, suppose a set of three actions: spam (S), not spam (N), uncertain (U). Which one should be picked? First, one has to define a cost function, which is represented in the following table

Actions	Is spam	Is not spam
S	0	10
N	1	0
U	5	5

In this case one can compute the expected cost of each action and pick the one with lower cost. In this case it holds:

- Cost of S: $(1-p)\cdot 10$
- Cost of N: p
- Cost of U: $p\cdot 5 + (1-p)\cdot 5$

Example 6.3.2 (Optimal Decision for Logistic Regression) In the example of logistic regression for binary classification we have:

- Estimated conditional distribution: $\hat{\mathbb{P}}\left[y|\mathbf{x}\right] = Ber(y;\sigma(\mathbf{w}^T\mathbf{x}))$
- Action set: $\mathcal{A} = \{+1,-1\}$
- Cost function: $C(y,a) = \left[y \neq a\right]$

Then the action that minimizes the expected cost is the most likely class.

Example 6.3.3 (Asymmetric Cost) Consider the following situation:

- Estimated conditional distribution: $\hat{\mathbb{P}}\left[y|\mathbf{x}\right] = Ber(y;\sigma(\mathbf{w}^T\mathbf{x}))$
- Action set: $\mathcal{A} = \{+1,-1\}$
- Costs:

$$C(y,a) = \begin{cases} c_{FP} & \text{if } y=-1 \text{ and } a=+1 \\ c_{FN} & \text{if } y=1 \text{ and } a=-1 \\ 0 & \text{otherwise} \end{cases}$$

Then the expected costs for our set of actions are:

- $c_+ = (1-p)\cdot c_{FP}$
- $c_- = p\cdot c_{FN}$

and we have to pick the smallest one, *i.e.* we predict +1 if and only if $p > \frac{c_{FP}}{c_{FP}+c_{FN}}$

Example 6.3.4 (Asymmetric Cost for Regression) Consider the following situation:

- Estimated conditional distribution: $\hat{\mathbb{P}}\left[y|\mathbf{x}\right] = \mathcal{N}(y;\mathbf{w}^T\mathbf{x},\sigma^2)$
- Action set: $\mathcal{A} = \mathbb{R}$

- Costs: $C(y, a) = c_1 \max(y - a, 0) + c_2 \max(a = y, 0)$

This means, that underestimating and overestimating have different costs. Then the action that minimizes the expected cost is

$$a^* = \hat{\mathbf{w}}^T \mathbf{x} + \sigma \Phi^{-1}(\frac{c_1}{c_1 + c_2})$$

Example 6.3.5 (Doubtful Logistic Regression) Consider the following situation:

- Estimated conditional distribution: $\hat{\mathbb{P}}\left[y|\mathbf{x}\right] = Ber(y; \sigma(\mathbf{w}^T \mathbf{x}))$
- Action set: $\mathcal{A} = \{+1, -1, D\}$
- Costs:

$$C(y, a) = \begin{cases} \left[y \neq a\right] & \text{if } a \in \{+1, -1\} \\ c & \text{if } a = D \end{cases}$$

Then the action that minimizes the expected cost is given by

$$a^* = \begin{cases} y & \text{if } \hat{\mathbb{P}}\left[y|\mathbf{x}\right] \geq 1 - c \\ D & \text{otherwise} \end{cases}$$

That is, we pick the most likely class only if confident enough.

In Machine Learning there is a golden (business) rule: *a labelled dataset is money*. In facts, obtaining data is relatively cheap, but obtaining labels is more expensive. Hence minimizing the number of labels is a useful goal.

Definition 6.3.2 (Active Learning) *Active learning is a technique that uses algorithms in order to minimize the number of labels. A simple strategy is **uncertainty sampling**, which follows the principle of always picking the most uncertain example.*

Algorithm 6.1: Uncertainty Sampling

```
Given: pool of unlabeled examples D_X = {x_1, ..., x_n}
Also maintain a labelled dataset D, initially empty
for t = 1, 2, 3, ...
    Estimate P̂[Y_i|x_i] given current data D
    i_t ∈ arg min_i |0.5 − P̂[Y_i|x_i]|
    Query label for y_{i_t} and D ← D ∪ {(x_{i_t}, y_{i_t})}
end
```

6.4 Generative Modeling

As a motivational example think to the scenario of classification. If we want to classify a feature vector in the negative region which is far away from the boundary, a classification method such as linear regression would be very confident to predict that the point has a negative label. This also if the new point that has to be classified is an *outlier*, *i.e.* far away from all other negative examples that the model has seen so far.

This means that logistic regression can be overconfident about labels for outliers.

So far, we have considered learning methods that estimate conditional distributions $\mathbb{P}\left[y|\mathbf{x}\right]$. Such models don't attempt to esimate $\mathbb{P}\left[\mathbf{x}\right]$ and thus they will not be able to detect outliers, *i.e.* unusual points for which $\mathbb{P}\left[\mathbf{x}\right]$ is very small. Thus models are called *discriminative models*. Now we consider the so called *generative models* that aim to estimate the joint distribution $\mathbb{P}\left[\mathbf{x}, y\right]$. Keep in mind that generative models are more powerful than discriminative models, in facts it is possible to derive $\mathbb{P}\left[y|\mathbf{x}\right]$ from $\mathbb{P}\left[\mathbf{x}, y\right]$ (we have $\mathbb{P}\left[y|\mathbf{x}\right] = \frac{\mathbb{P}\left[\mathbf{x},y\right]}{\mathbb{P}\left[\mathbf{x}\right]}$) but not viceversa.

The typical approach to generative modeling is attempting to infer the process, according to which examples are generated and then do the following:

- Estimate prior on labels $\mathbb{P}\left[y\right]$
- Estimate conditional distribution $\mathbb{P}\left[\mathbf{x}|y\right]$ for each class y
- Obtain predictive distribution using Bayes'rule

$$\mathbb{P}\left[y|\mathbf{x}\right] = \frac{1}{\mathbb{P}\left[\mathbf{x}\right]}\mathbb{P}\left[y\right]\mathbb{P}\left[\mathbf{x}|y\right]$$

Usually, in the family of Naive classifiers, features are assumed to be conditionally independent given Y, *i.e.* $\mathbb{P}\left[\mathbf{X_1}, \ldots, \mathbf{X_d}|\mathbf{Y}\right] = \prod_{i=1}^{d}\mathbb{P}\left[\mathbf{X_i}|\mathbf{Y}\right]$.

Example 6.4.1 (Gaussian Naive Bayes Classifier) Learning: given data $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$.

- MLE for class prior: $\hat{\mathbb{P}}\left[Y = y\right] = \frac{\text{Count}(Y=y)}{n}$
- MLE for feature distribution $\hat{\mathbb{P}}\left[\mathbf{x}_i|y_i\right] = \mathcal{N}(\mathbf{x}_i; \hat{\mu}_{y,i}, \sigma^2_{y,i})$

$$\hat{\mu}_{y,i} = \frac{1}{\text{Count}(Y = y)}\sum_{j:y_j=y} x_{j,i}$$

$$\sigma^2_{y,i} = \frac{1}{\text{Count}(Y = y)}\sum_{j:y_j=y}(x_{j,i} - \hat{\mu}_{y,i})^2$$

Prediction given new point $\mathbf{x}$

$$y = \arg\max_{y'}\hat{\mathbb{P}}\left[y'|\mathbf{x}\right] = \arg\max_{y'}\hat{\mathbb{P}}\left[y'\right]\sum_{i=1}^{d}\hat{\mathbb{P}}\left[\mathbf{x}_i|y_i\right]$$

Here one could show that, in the case of binary classification and with the additional assumption of shared variance, the Gaussian Naive Bayes Classifier produces a linear classifier of the same form as logistic regression. For the sake of brevity we omit this argument and we point to the literature.

This model has some limitations, such as the fact that if the conditional independence assumption is violated (*i.e.* features are not generated independently) then the predictions might become overconfident. This might be fine if we are interested in the most likely outcome only, but this would hurt if we use this probability to make decisions. In order

to improve this issue, we introduce a new (more complex) model in the next example.

Example 6.4.2 (Gaussian Bayes Classifier) Learning: given data $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$.

- MLE for class prior: $\hat{\mathbb{P}}\left[Y = y\right] = \frac{\text{Count}(Y=y)}{n}$
- MLE for feature distribution $\hat{\mathbb{P}}\left[\mathbf{x}|y\right] = \mathcal{N}(\mathbf{x}; \hat{\mu}_y, \hat{\Sigma}_y)$

$$\hat{\mu}_y = \frac{1}{\text{Count}(Y = y)} \sum_{i:y_i=y} x_i$$

$$\hat{\Sigma}_y = \frac{1}{\text{Count}(Y = y)} \sum_{i:y_i=y} (\mathbf{x}_i - \hat{mu}_y)(\mathbf{x}_i - \hat{mu}_y)^T$$

Prediction given new point $\mathbf{x}$

$$y = \arg\max_{y'} \hat{\mathbb{P}}\left[y'|\mathbf{x}\right] = \arg\max_{y'} \hat{\mathbb{P}}\left[y'\right] \sum_{i=1}^{d} \hat{\mathbb{P}}\left[\mathbf{x}_i|y_i\right]$$

Given $p := \mathbb{P}[Y = 1]$ and $\mathbb{P}\left[\mathbf{x}|y\right] = \mathcal{N}(\mathbf{x}; \mu_y, \Sigma_y)$ we want to compute the discriminant

$$f(\mathbf{x}) = \log \frac{\mathbb{P}[Y = 1|\mathbf{x}]}{\mathbb{P}[Y = 1|\mathbf{x}]}$$

This discriminant function is given by

$$f(\mathbf{x}) = \log \frac{p}{1-p} + \frac{1}{2}\left[\log \frac{|\hat{\Sigma}_-|}{|\hat{\Sigma}_+|} + \left((\mathbf{x} - \hat{\mu}_-)\hat{\Sigma}_-^{-1}(\mathbf{x} - \hat{\mu}_-)\right) - \left((\mathbf{x} - \hat{\mu}_+)\hat{\Sigma}_+^{-1}(\mathbf{x} - \hat{\mu}_+)\right)\right]$$

By fixing $p = 0.5$ and with the additional assumption $\hat{\Sigma}_- = \hat{\Sigma}_+$, then one obtains a linear classifier known as *Fisher's linear discriminant analysis* which, as happened with the Naive Gaussian Bayes Classifier, has the same class distribution of logistic regression. Without those further assumptions, we do *quadratic discriminant analysis*.

We have introduced generative modeling which is in contrast with the discriminative models we had discussed before. This introduces some trade-offs:

- Fisher's Linear Discriminant Analysys
  - Is a generative model, *i.e.* it models $\mathbb{P}[\mathbf{X}, Y]$
  - Can be used to detect outliers, *i.e.* $\mathbb{P}[\mathbf{X}]$ is small
  - Assumes normality of $\mathbf{X}$
  - Not very robust against violation of this assumption
- Logistic regression
  - Discriminative model, *i.e.* models $\mathbb{P}[Y|\mathbf{X}]$ only
  - Cannot detect outliers
  - Makes no assumptions on $\mathbb{R}X$
  - More robust

Moreover, in the class of generative models, we have a naive classifier which assumed conditional independence of features and a more involved

one which considered covariances. Also in this situation we have some trade-offs:

- Naive Gaussian Bayes Models
  - Conditional independence assumption may lead to overconfidence
  - Predictions might still be useful
  - The number of parameters is in $\mathcal{O}(cd)$
  - Complexity (memory and inference) is linear in d
- General Gaussian Bayes Models
  - Captures correlations among features
  - Avoids overconfidence
  - The number of parameters is in $\mathcal{O}(cd^2)$
  - Complexity is quadratic in d

Unsupervised Learning

Classification 7

7.1 Clustering 73
7.2 K-Means Clustering 74

7.1 Clustering

In unsupervised clustering we are given a set of features vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$, where $\mathbf{x}_i \in \mathbb{R}^d$, and the goal is to group the given data points into clusters such that *similar* points are in the same cluster and *dissimilar* points are in different clusters. Different clustering approaches define their own metric to describe the similarity between two points.

Applications

- *Words clustering*: Given a document, group the words based on what they describe.
- *Image Clustering*: Given a set of images, group them based on their features.
- *Outlier Detection*: Given a set of vectors, group them to find which one are outliers.
- Given a set of products, group them based on which type of customer bought them.

Clustering Approaches

- *Hierarchical Clustering* separates the data points into small clusters by distance (norm), then the small clusters are again separated in coarser and coarser clusters until all the points are in one big cluster (this way is bottom-up but could also be done top-down). By representing each group of clusters with a node and connecting sub-clusters with parent clusters we can represent the entire structure as a hierarchical tree. Then by chopping the branches of the structure at different heights we can get many small or few big clusters. Some algorithms are single/average-linkage clustering.
- *Partitional Clustering* Partitional clustering uses a graph data structure to connect data points depending on some cost function. Then using different of graph cuts (*e.g.* min-cut) we get different clusters. Some algorithms are spectral clustering or graph-cut based clustering.
- *Model-Based Clustering* We represent each cluster by a model (*e.g.* the center, which means that we will assign to each point the closest center), then for new points, we will infer the cluster by picking which model fits best. Some algorithms are k-means clustering or Gaussian mixture models.

Model-based clustering has the advantage that given a new unseen data point we can easily apply the model and infer to which cluster it should be part of. In hierarchical/partitional clustering we apply the structure only on points that are already given and hence it's less flexible. More specifically we will look into *k-means* clustering.

7.2 K-Means Clustering

Goal

Given a set of feature vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ where $\mathbf{x}_i \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$), and a desired number of output clusters $k \in \mathbb{N}$.

Output a set of cluster centers $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ where $\boldsymbol{\mu}_i \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{M} \in \mathbb{R}^{k \times d}$) such that the empirical error

$$\hat{R}(\mathbf{M}) = \hat{R}(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k) := \sum_{i=1}^{n} \min_{j \in \{1,\ldots,k\}} \left\| \mathbf{x}_i - \boldsymbol{\mu}_j \right\|_2^2 \tag{7.1}$$

is minimal, *i.e.* find $\hat{\mathbf{M}}$ such that

$$\hat{\mathbf{M}} = \arg\min_{\mathbf{M}} \hat{R}(\mathbf{M}) \tag{7.2}$$

In simpler terms, we want to find k cluster centers ($\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$), such that the squared distance between the cluster centers (which is how we define the similarity) and all the points is minimal. Then given a new unseen point $\mathbf{x}$ we will put it into the cluster with the closest center $\boldsymbol{\mu}_j$.

The problem with this approach is that $\hat{R}$ is a non-convex function (because of the min operator), and thus this optimization problem is NP-hard (*i.e.* can't be solved optimally). One solution is to use gradient descent, however other than selecting the initial values, which if not picked correctly might give us a bad solution, we would also have to manually select the learning rate. A better solution is to use *Lloyd's algorithm*.

Lloyd's algorithm is an iterative algorithm that is guaranteed to monotonically decrease at each step and hence it will always converge to a local optimum. The idea is that we initialize k random centers μ_j, then for each point $\mathbf{x}_i$, at each iteration, we find the closest center index $z_i \in \{1, \ldots, k\}$ and update all previous means $\boldsymbol{\mu}_j$ to be the average point of all $\mathbf{x}_i$ that have j as their closest center (*i.e.* $z_i = j$).

Algorithm 7.1: Lloyd's Algorithm

$\mathbf{M}^{(0)} = [\boldsymbol{\mu}_1^{(0)}, \ldots, \boldsymbol{\mu}_k^{(0)}]$ ▷ *Initialize cluster centers.*
$t \leftarrow 1$
while not converged
  $z_i^{(t)} \leftarrow \arg\min_{j \in \{1,\ldots,k\}} \left\| \mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)} \right\|_2^2$ ▷ *Assign each $\mathbf{x}_i$ to the closest center.*
  $\boldsymbol{\mu}_j^{(t)} \leftarrow \frac{1}{n_j} \sum_{i : z_i^{(t)} = j} \mathbf{x}_i$ ▷ *Set new center as* mean *of assigned points.*
  t++
end
return $\mathbf{M}^{(t)} = [\boldsymbol{\mu}_1^{(t)}, \ldots, \boldsymbol{\mu}_k^{(t)}]$

Where n_j is the number of points in cluster j (*i.e.* centered at $\boldsymbol{\mu}_j$), and $i : z_i^{(t)} = j$ means all i such that $z_i^{(t)} = j$. Each step has a computational

complexity of $\mathcal{O}(ndk)$.

Lloyd's algorithm is guaranteed to find a local minimum since at each step it decreases monotonically.

Lemma 7.2.1 (Lloyd's Monotonic Decrease) *Let* $z_i^{(t)} \in \{1, \dots, k\}$ *be the index of the closest center* $\boldsymbol{\mu}_{z_i^{(t)}}$ *of vector* $\mathbf{x}_i$ *at step* t *and* $\hat{R}\left(\boldsymbol{\mu}, z\right) := \sum_{i=1}^{n} \left\|\mathbf{x_i} - \boldsymbol{\mu}_{z_i}\right\|$ *be a single center error, then:*

$$\hat{R}\left(\boldsymbol{\mu}^{(t)}, z^{(t)}\right) \geq \hat{R}\left(\boldsymbol{\mu}^{(t+1)}, z^{(t+1)}\right) \tag{7.3}$$

Proof.

$$\hat{R}\left(\boldsymbol{\mu}^{(t)}, z^{(t)}\right) \geq \hat{R}\left(\boldsymbol{\mu}^{(t)}, z^{(t+1)}\right), \qquad z^{(t+1)} = \arg\min_{z} \hat{R}(\boldsymbol{\mu}^{(t)}, z) \tag{7.4}$$

$$\geq \hat{R}\left(\boldsymbol{\mu}^{(t+1)}, z^{(t+1)}\right), \quad \boldsymbol{\mu}^{(t+1)} = \arg\min_{\mu} \hat{R}(\boldsymbol{\mu}, z^{(t+1)}) \tag{7.5}$$

$$\tag{7.6}$$

□

However, there are still a few problems:

- *Exponential iterations*: even if the algorithm is guaranteed to find a local optimum it might take an exponential number of steps. This problem manifests itself only in some rare cases, and hence can be usually ignored. In fact, the number of steps to convergence is usually very small.
- *Initialization*: how to initialize the centers $\boldsymbol{\mu}_j^{(0)}$? We have seen that it's guaranteed to converge but usually to local optima, what if the local optima is really bad? This problem heavily depends on initialization.
- *Cluster number*: how do we pick the number of clusters k if we don't know in how many clusters can our data be separated?
- *Cluster shape*: we represent a cluster by a central point, however, it's not always the case that a cluster can be represented by a single mean point. This can be solved with *kernel-k-means* clustering, similarly to how we use kernels in supervised learning to fit non-linear functions.

Initialization approaches

- *Random Start*: we can k points among $\mathbf{x}_i$ and set them as our initial $\boldsymbol{\mu}_j$. However, if there are some large clusters and some small clusters the probability of picking a point in the large cluster is much higher and thus we might find a bad solution.
- *Farthest Points Heuristic*: instead of picking k random points among $\mathbf{x}_i$ we pick one center at the time and for each new point if it's further away than the other centers it will have a higher probability of being selected. This approaches works really well if our data doesn't contain outliers, however if it does it will pick outliers with a high probability and thus fail to find a good solution.

- *K-Means++*: is a variant of the point heuristic, where instead of only considering points that are far away from the current centers (to solve the problem of picking points in the same cluster), we also increase the probability of picking points in clusters that have many points (to solve the problem of picking outlier points). K-Means++ is usually used as the standard initializer.

Algorithm 7.2: K-Means++ Initialization

$i_1 \sim \text{Uniform}(\{1, \dots, n\})$ ▷ *Pick first center randomly.*
$\boldsymbol{\mu}_1^{(0)} \leftarrow \mathbf{x}_{i_1}$
for $j = 2, \dots, k$
  pick i_j with probability $\frac{1}{z} \min_{l \in \{1,\dots,j-1\}} \left\| \mathbf{x}_{i_j} - \boldsymbol{\mu}_l^{(0)} \right\|_2^2$
  $\boldsymbol{\mu}_j^{(0)} = \mathbf{x}_{i_j}$
end
return $\mathbf{M}^{(0)} = [\boldsymbol{\mu}_1^{(0)}, \dots, \boldsymbol{\mu}_k^{(0)}]$

This initialization technique, other than picking the initial points, already gives us a good guess for the optimum without using Lloyd's algorithm or other model-based clustering algorithms.

Lemma 7.2.2 (K-Means++ Log Competitive) *If we pick* $\mathbf{M}^{(0)}$ *as our final guess for the centers, assuming that* $\mathbf{M}^{(0)}$ *is sampled from a random process, then:*

$$\mathbb{E}[\hat{R}(\mathbf{M}^{(0)})] \leq \mathcal{O}(\log(k)) \min_{\mathbf{M}} \hat{R}(M) \tag{7.7}$$

In simpler terms, if we pick $\mathbf{M}^{(0)}$ as our final guess it's only a logarithm term away from the optimal k-means solution.

Cluster Number Selection In general, picking the number of clusters k is very difficult. With supervised learning we know the type of model that we want to fit. In unsupervised clustering we don't really know. This is still an unsolved problem, and the standard approaches usually are:

- *Elbow Method Heuristic*: we start with a low value of k and then increase it until the error $\hat{R}(\mathbf{M})$ decreases by only a negligible amount, then pick the second-last guess (if you graph the error the second-last guess is pointy and looks like an elbow.
- *Regularization*: instead of minimizing only $\hat{R}(\mathbf{M})$ we add a regularization term weighted as always by λ.

$$\min_{k, \mathbf{M}_{1:k}} \hat{R}(\mathbf{M}_{1:k}) + \lambda k \tag{7.8}$$

  This is in fact equivalent to the elbow method.

Note that we can't use cross-validation because by increasing k the error decreases continuously until $k = n$, where the error will be 0. But a smaller error with a large k is not what we are looking for, we want both the error and k to be as small as possible.

8 Regression

8.1 Dimension Reduction 77
8.2 Principal Component Analysis (PCA) 77
8.3 Kernel PCA 82
8.4 Autoencoders 84

8.1 Dimension Reduction

Dimension reduction is a method that given a set of vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ where $\mathbf{x}_i \in \mathbb{R}^d$, returns some other vectors $\mathbf{z}_1, \dots, \mathbf{z}_n$ where $\mathbf{z}_i \in \mathbb{R}^k$ with $k < d$. In other words, we want to represent the same amount of points/vectors but in a smaller dimension without loosing too much information. This method has many applications:

- *Visualization*: we can't visualize vectors in more than 3 dimensions, dimension reduction helps to shrink the dimension of the data to 3 or less and hence give us a visible or intuitive understanding.
- *Regularization*: shrinking the dimension of the data without losing too much information is a form of regularization. If we give as input to a model lower-dimensional data it will train faster and be able to find the important characteristics with less work (*i.e.* reduce the model complexity).
- *Unsupervised Feature Discovery*: similar to the point above by shrinking the dimension the feature that are most important will remain encoded in the vectors.

8.2 Principal Component Analysis (PCA)

Goal

Given a set of centered[a] feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ where $\mathbf{x}_i \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$), and the desired output dimension $k \in \mathbb{N}$ with $1 \leq k \leq d$

Output the analog set of dimensionally reduced vectors $\mathbf{z}_1, \dots, \mathbf{z}_n$ where $\mathbf{z}_i \in \mathbb{R}^k$ (which can be represented as a matrix $\mathbf{Z} \in \mathbb{R}^{n \times k}$) and an orthogonal matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$ such that the empirical error

$$\hat{R}(\mathbf{W}, \mathbf{Z}) = \hat{R}(\mathbf{W}, \mathbf{z}_1, \dots, \mathbf{z}_n) \coloneqq \sum_{i=1}^{n} \|\mathbf{W}\mathbf{z}_i - \mathbf{x}_i\|_2^2 \tag{8.1}$$

is minimal, i.e. find $(\hat{\mathbf{W}}, \hat{\mathbf{Z}})$ such that:

$$(\hat{\mathbf{W}}, \hat{\mathbf{Z}}) = \arg\min_{\mathbf{W}, \mathbf{Z}} \hat{R}(\mathbf{W}, \mathbf{Z}) \tag{8.2}$$

[a] $\boldsymbol{\mu} \coloneqq \sum_{i=1}^{n} \mathbf{x}_i = \mathbf{0}$

The reason why we output not only the dimensionally reduced vectors in $\mathbf{Z}$, but also a matrix $\mathbf{W}$ is that given a new point $\mathbf{x}$ that is not in the initial dataset we can easily find the analog dimensionally reduced vector $\mathbf{z}$ by only computing $\mathbf{z} \coloneqq \mathbf{W}^T\mathbf{x}$.[1] In other words $\mathbf{W}^T$ is a transformation matrix

1: The coefficients $\mathbf{z}_i \coloneqq \mathbf{W}^T\mathbf{x}_i$ of the projected vector $\mathbf{x}_i$ are called *principal scores*.

from d dimensional coordinates to k dimensional coordinates, and $\mathbf{W}$ is a transformation matrix from k dimensional coordinates to d dimensional coordinates. Since the transformation $\mathbf{W}$ is trying to reconstruct higher dimensional vectors $\mathbf{x}$ from compressed lower dimensional vectors $\mathbf{z}$ we *must* loose some information (except if $k = d$, then we would reconstruct the exact same data). If we let $\bar{\mathbf{x}}_i := \mathbf{W}\mathbf{z}_i$ be the reconstructed version of $\mathbf{x}_i$, the goal of PCA is to loose as little information as possible, *i.e.* make sure that $\mathbf{x}_i$ is as close as possible to $\bar{\mathbf{x}}_i$. More precisely, we want to find a matrix $\mathbf{W}$ such that when we take the compressed version of $\mathbf{x_i}$, $\mathbf{z}_i$, we can reconstruct the initial $\mathbf{x}_i$ with the least possible error, that is, minimize the reconstruction error $\|\bar{\mathbf{x}}_i - \mathbf{x}_i\|_2^2 = \|\mathbf{W}\mathbf{z}_i - \mathbf{x}_i\|_2^2$ for all i. As always we have defined the similarity/distance of two vectors as the squared norm.

The nice thing about PCA is that we can find the *globally* optimal solution $(\mathbf{W}^*, \mathbf{Z}^*)$ by only using ideas from linear algebra, in facts we only have to find $\mathbf{W}^*$, then $\mathbf{Z}^* := (\mathbf{W}^*)^T\mathbf{X}^T$. So the idea is to somehow compute $\mathbf{W}^*$ using $\mathbf{X}$, and then compute $\mathbf{Z}^*$ as described.

Minimizing Error to Maximizing Variance Before computing $\mathbf{W}^*$ we will show how to convert the problem of minimizing the reconstruction error into a maximization problem.

Lemma 8.2.1 (Min Max PCA) *Let the matrix* $\mathbf{W}^*$ *be*

$$\mathbf{W}^* = \begin{bmatrix} | & | & & | \\ \mathbf{w}_1^* & \mathbf{w}_2^* & \dots & \mathbf{w}_k^* \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{d\times k} \tag{8.3}$$

then to find $\mathbf{w}_j^* \in \mathbb{R}^d$*, which is called the* j*-th* principal component/axis, *we can solve either of the following two optimization problem:*

$$\begin{aligned} \mathbf{w}_j^* &= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j \\ \mathbf{z}_1,\dots,\mathbf{z}_n}}{\arg\min} \sum_{i=1}^n \left\|\mathbf{w}_j z_{i,j} - \mathbf{x}_i\right\|_2^2 && (8.4) \\ &= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\max} \mathbf{w}_j^T\mathbf{X}^T\mathbf{X}\mathbf{w}_j && (8.5) \end{aligned}$$

Proof.

$$\begin{aligned} \mathbf{w}_j^* &= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j \\ \mathbf{z}_1,\dots,\mathbf{z}_n}}{\arg\min} \sum_{i=1}^n \left\|\mathbf{w}_j z_{i,j} - \mathbf{x}_i\right\|_2^2 && (8.6) \\ &= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\min} \sum_{i=1}^n \left\|\mathbf{w}_j\mathbf{w}_j^T\mathbf{x}_i - \mathbf{x}_i\right\|_2^2, \quad \textit{Def. } z_{i,j} := \mathbf{w}_j^T\mathbf{x}_i && (8.7) \\ &= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\min} \sum_{i=1}^n (\mathbf{w}_j\mathbf{w}_j^T\mathbf{x}_i - \mathbf{x}_i)^T(\mathbf{w}_j\mathbf{w}_j^T\mathbf{x}_i - \mathbf{x}_i), \quad \textit{Def. L2} && (8.8) \end{aligned}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\min} \sum_{i=1}^{n}(\mathbf{x}_i^T\mathbf{w}_j\mathbf{w}_j^T - \mathbf{x}_i^T)(\mathbf{w}_j\mathbf{w}_j^T\mathbf{x}_i - \mathbf{x}_i) \tag{8.9}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\min} \sum_{i=1}^{n}\mathbf{x}_i^T\mathbf{w}_j\underbrace{\mathbf{w}_j^T\mathbf{w}_j}_{=1}\mathbf{w}_j^T\mathbf{x}_i - 2\mathbf{x}_i^T\mathbf{w}_j\mathbf{w}_j^T\mathbf{x}_i + \underbrace{\mathbf{x}_j^T\mathbf{x}_i}_{\|\mathbf{x}_i\|_2^2} \tag{8.10}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\min} \sum_{i=1}^{n} -\mathbf{x}_i^T\mathbf{w}_j\mathbf{w}_j^T\mathbf{x}_i + \|\mathbf{x}_i\|_2^2 \tag{8.11}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\min} \underbrace{\sum_{i=1}^{n}\|\mathbf{x}_i\|_2^2}_{Const.} - \sum_{i=1}^{n}(\mathbf{w}_j^T\mathbf{x}_i)^2 \tag{8.12}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\max} \sum_{i=1}^{n}(\mathbf{w}_j^T\mathbf{x}_i)^2, \quad \textit{Min to max by switching sign.} \tag{8.13}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\max} \sum_{i=1}^{n}\mathbf{w}_j^T\mathbf{x}_i\mathbf{x}_i^T\mathbf{w}_j \tag{8.14}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\max} \mathbf{w}_j^T\left(\sum_{i=1}^{n}\mathbf{x}_i\mathbf{x}_i^T\right)\mathbf{w}_j \tag{8.15}$$

$$= \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\max} \mathbf{w}_j^T\mathbf{X}^T\mathbf{X}\mathbf{w}_j \tag{8.16}$$

□

The first minimization problem has the same form as our initial optimization problem, where instead of solving directly for $\mathbf{W}^*$ we solve for each component individually. We add two additional constraints, the first constraint $\mathbf{w}_j^T\mathbf{w}_j = \|\mathbf{w}_j\|_2 = 1$ is to make sure that our principal axis has length one such that our solution is unique, the second constraint $\mathbf{w}_j^T\mathbf{w}_l = 0$ is called orthogonality constraint, to make sure that all principal axis are orthogonal between each other. Lastly, note that by converting the problem into a maximization problem we can discard the dependence with $\mathbf{Z}$ and hence the second form is less constrained.

This min/max duality has a nice geometric interpretation. The variance of the centered points $\mathbf{x}_i$ projected on a unit vector $\mathbf{w}_j$ is given by $\frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i^T\mathbf{w}_j)^2$ which is the same as in 8.13 (up to an irrelevant factor of $\frac{1}{n}$ which doesn't affect the optimization), thus finding the unit vector $\mathbf{w}_j$ that *maximizes* the variance of the projected points[2] is the same as finding the unit vector that *minimizes* the reconstruction error of the projected points.

Finding Optimum Now that we have an easier form of the optimization problem we can finally find $\mathbf{W}^*$ by using the following lemma.

2: Assume that $\mathbf{z}_1, \dots, \mathbf{z}_n$ are i.i.d. random observations from a random variable $\mathbf{Z}$, then $\mathbb{E}[\mathbf{Z}] = 0$ since the points are centered and by the Law of large numbers:

$$\mathbb{V}[\mathbf{Z}] = \mathbb{E}[\mathbf{Z}^2] \overset{LLN}{\approx} \frac{1}{n}\sum_{i=1}^{n}\mathbf{z}_i^2 \tag{8.17}$$

$$= \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i^T\mathbf{w}_j)^2 \tag{8.18}$$

Lemma 8.2.2 (PCA) *Let $\mathbf{X}^{n\times d}$ be the feature matrix, $j \in \{1,\dots,k\}$, and $\mathbf{v}_j$ be the eigenvector associated to the j-th* largest[a] *eigenvalue[b] λ_j of $\mathbf{\Sigma} \coloneqq \frac{1}{n}\mathbf{X}^T\mathbf{X}$ (i.e. $\mathbf{\Sigma}\mathbf{v}_j = \lambda_j\mathbf{v}_j$)[c], then $\mathbf{w}_j^* = \mathbf{v}_j$ with*

$$\mathbf{w}_j^* = \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0,\forall 1\le l<j}}{\arg\max} \mathbf{w}_j^T\mathbf{X}^T\mathbf{X}\mathbf{w}_j \tag{8.19}$$

i.e. $\mathbf{v}_j$ is equal to the j-th pricipal component $\mathbf{w}_j^*$.

[a] $(\lambda_1 \ge \cdots \ge \lambda_k \ge \cdots \ge \lambda_d)$
[b] λ_j is called *principal eigenvalue* of the j-th principal component.
[c] The matrix $\mathbf{\Sigma}$ is called covariance matrix of $\mathbf{X}$, note that this definition is true only if the rows $\mathbf{x}_i$ are centered.

Proof. We will prove that for all $j \in \{1,\dots,k\}$ the lemma holds by induction on j.
Base Case ($j = 1$): note that for $j = 1$ there is no orthogonality constraint.

$$\mathbf{w}_1^* = \underset{\mathbf{w}_j^T\mathbf{w}_j=1}{\arg\max}\, \mathbf{w}_1^T\mathbf{X}^T\mathbf{X}\mathbf{w}_1 \tag{8.20}$$

$$= \underset{\|\mathbf{w}_1\|_2=1}{\arg\max}\, \mathbf{w}_1^T \underbrace{n}_{Const.} \mathbf{\Sigma}\mathbf{w}_1, \quad Def.\ \mathbf{\Sigma} \tag{8.21}$$

$$= \underset{\|\mathbf{w}_1\|_2=1}{\arg\max}\, \mathbf{w}_1^T\mathbf{\Sigma}\mathbf{w}_1 \tag{8.22}$$

$$= \underset{\|\mathbf{w}_1\|_2=1}{\arg\max}\, \mathbf{w}_1^T\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T\mathbf{w}_1, \quad Eigendecomposition \tag{8.23}$$

$$= \underset{\|\mathbf{u}_1\|_2=1}{\arg\max}\, \mathbf{u}_1^T\mathbf{\Lambda}\mathbf{u}_1, \quad Let\ \mathbf{u}_j \coloneqq \mathbf{V}^T\mathbf{w}_j \tag{8.24}$$

$$= \underset{\|\mathbf{u}_1\|_2=1}{\arg\max} \sum_{i=1}^{d} \lambda_i u_{1,i}^2 \tag{8.25}$$

$$\tag{8.26}$$

Note that $\mathbf{\Sigma} \coloneqq \frac{1}{n}\mathbf{X}^T\mathbf{X}$ is a symmetric and positive definite matrix and thus has an eigendecomposition of the form $\mathbf{\Sigma} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$, where $\mathbf{V} \in \mathbb{R}^{d\times d}$ is orthonormal *i.e.* $\mathbf{V}^T\mathbf{V} = \mathbf{V}\mathbf{V}^T = \mathbf{I}$ and $\mathbf{\Lambda} = \mathrm{diag}(\lambda_1,\dots,\lambda_d)$ with $\lambda_1 \ge \cdots \ge \lambda_d$. Furthermore $\mathbf{V}$ contains the eigenvectors of $\mathbf{\Sigma}$ as columns. Note that since $\mathbf{V}$ is orthonormal we have that $\|\mathbf{u}_1\|_2 = \left\|\mathbf{V}^T\mathbf{w}_1\right\|_2 = \|\mathbf{w}_1\|_2 = 1$. Finally we have to pick a *unit* vector $\mathbf{u}_1$ that maximizes the sum $\sum_{i=1}^d \lambda_i u_{1,i}^2$, since the eigenvalues λ_i are sorted from largest (λ_1) to the smallest (λ_d) the best we can do is set $\mathbf{u}_1^* = \mathbf{e}_1$, where $\mathbf{e}_1 = [1,0,\dots,0]^T \in \mathbb{R}^d$ (the first unit vector) such that our sum will equal λ_1 *i.e.* be as big as possible. Then to find the optimal $\mathbf{w}_1^*$:

$$\mathbf{u}_j^* \coloneqq \mathbf{V}^T\mathbf{w}_j^* \Leftrightarrow \mathbf{V}\mathbf{u}_j^* = \overbrace{\mathbf{V}\mathbf{V}^T}^{=\mathbf{I}}\mathbf{w}_j^* \tag{8.27}$$

$$\Leftrightarrow \mathbf{V}\mathbf{u}_j^* = \mathbf{w}_j^* \tag{8.28}$$

$$\Leftrightarrow \mathbf{w}_1^* = \mathbf{V}\mathbf{u}_1 = \mathbf{V}\mathbf{e}_1 = \mathbf{v}_1 \tag{8.29}$$

$$\tag{8.30}$$

Where $\mathbf{v}_1$ is the eigenvector of $\mathbf{\Sigma}$ associated with the biggest eigenvalue

λ_1, which is what we wanted to prove.

Induction Hypothesis:

$$\mathbf{v}_j = \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0, \forall 1\leq l<j}}{\arg\max} \mathbf{w}_j^T\mathbf{X}^T\mathbf{X}\mathbf{w}_j \tag{8.31}$$

where $\mathbf{v}_j$ is the unit eigenvector associated to the j-th largest eigenvalue of $\mathbf{\Sigma}$.

Induction Step ($j \longrightarrow j+1$):
We will show that

$$\mathbf{v}_{j+1} = \underset{\substack{\mathbf{w}_{j+1}^T\mathbf{w}_{j+1}=1 \\ \mathbf{w}_{j+1}^T\mathbf{w}_l=0, \forall 1\leq l<j+1}}{\arg\max} \mathbf{w}_{j+1}^T\mathbf{X}^T\mathbf{X}\mathbf{w}_{j+1} \tag{8.32}$$

by using the lagrange multiplier of $\mathbf{v}_{k+1}$ parametrized by λ and η_i.

$$\mathcal{L}(\mathbf{v}_{k+1}) := \mathbf{v}_{k+1}^T\mathbf{X}^T\mathbf{X}\mathbf{v}_{k+1} + \underbrace{\lambda\mathbf{w}_{j+1}^T\mathbf{w}_{j+1} - 1}_{\textit{Unitary Constraint}} + \underbrace{\sum_{i=1}^{j}\eta_i\mathbf{v}_{j+1}^T\mathbf{v}_i}_{\textit{Orthogonality Constraints}} \tag{8.33}$$

$$\nabla\mathcal{L}(\mathbf{v}_{k+1}) = 2\mathbf{X}^T\mathbf{X}\mathbf{v}_{k+1} - 2\lambda\mathbf{v}_{k+1} + \sum_{i=1}^{j}\eta_i\mathbf{v}_i \tag{8.34}$$

$$\stackrel{!}{=} \mathbf{0} \tag{8.35}$$

$$\tag{8.36}$$

By I.H. we know that all $\mathbf{v}_l$ with $l < j$ are orthogonal (*i.e.* $\mathbf{v}_l^T\mathbf{v}_j = 0$), we have to prove that this also holds for $l < j+1$. Observe that if we can prove that $\eta_l = 0$ for all l then all the orthogonality constraint are 0 and hence are satisfied.

$$0 = \mathbf{v}_l^T\mathbf{0} \tag{8.37}$$

$$= 2\mathbf{v}_l^T\mathbf{X}^T\mathbf{X}\mathbf{v}_{k+1} - 2\lambda\underbrace{\mathbf{v}_l^T\mathbf{v}_{k+1}}_{=0} + \sum_{i=1}^{j}\eta_i\underbrace{\mathbf{v}_l^T\mathbf{v}_i}_{=1 \textit{ only if } l=i} \tag{8.38}$$

$$= 2\mathbf{v}_l^T\mathbf{X}^T\mathbf{X}\mathbf{v}_{k+1} + \eta_l \tag{8.39}$$

$$= 2(\mathbf{X}^T\mathbf{X}\mathbf{v}_l)^T\mathbf{v}_{k+1} + \eta_l \tag{8.40}$$

$$= 2(\lambda_l\mathbf{v}_l)^T\mathbf{v}_{k+1} + \eta_l \qquad \textit{Def. EV (IH).} \tag{8.41}$$

$$= 2\lambda_l\underbrace{\mathbf{v}_l^T\mathbf{v}_{k+1}}_{=0} + \eta_l \tag{8.42}$$

$$= \eta_l \tag{8.43}$$

Finally by plugging back $\eta_l = 0$ in the gradient of the lagrangian we get that:

$$2\mathbf{X}^T\mathbf{X}\mathbf{v}_{k+1} - 2\lambda\mathbf{v}_{k+1} = \mathbf{0} \Leftrightarrow \mathbf{X}^T\mathbf{X}\mathbf{v}_{k+1} = \lambda\mathbf{v}_{k+1} \tag{8.44}$$

$$\tag{8.45}$$

hence λ is by definition an eigenvalue of $\mathbf{X}^T\mathbf{X}$ with eigenvector $\mathbf{v}_{k+1}$. □

Computing Optimum We can usually solve this optimization problem in two different ways:

- *Eigendecomposition*: as we have seen we can compute the matrix $\mathbf{\Sigma} := \frac{1}{n}\mathbf{X}^T\mathbf{X}$ and then applying eigendecomposition on $\mathbf{\Sigma}$ to get $\mathbf{\Sigma} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$. Then we have to set $\boldsymbol{W}^* = \mathbf{V}$.
- *Singular Value Decomposition*: has the same effect of eigendecomposition but the computation is more direct since we don't have to compute the matrix $\mathbf{\Sigma}$. By applying singular value decomposition on $\mathbf{X}$ we get:

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T \tag{8.46}$$

  where $\mathbf{U} \in \mathbb{R}^{n\times n}$ contains the eigenvectors of $\mathbf{X}\mathbf{X}^T$ in its columns, $\mathbf{V} \in \mathbb{R}^{d\times d}$ contains the eigenvectors of $\mathbf{X}^T\mathbf{X}$ in its columns, and $\mathbf{S} = \text{diag}(\sigma_1, \ldots, \sigma_d) \in \mathbb{R}^{n\times d}$ with $\sigma_1 \geq \cdots \geq \sigma_d$ where $\sigma_i^2 = \lambda_i$ and λ_i is the i-th largest eigenvalue of $\mathbf{X}$. Since $\mathbf{V}$ already contains the sorted eigenvectors of $\mathbf{X}^T\mathbf{X}$ we just have to set $\mathbf{W}^* = \mathbf{V}$.

Dimension Selection Choosing the value of k depends on the application.

- *Visualization*: to visualize the data clearly we can only pick $k \in \{1, 2, 3\}$.
- *Feature Induction*: if the dimension reduced features are given as input to a supervised learning algorithm we can use cross validation on k to find which one performs the best.
- *Elbow Method Heuristic*: we can start with a low k and then increase it until most of the variance of the data is accounted by the principal components.

8.3 Kernel PCA

If our feature vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are non-linearly separable and we use PCA to project our d dimensional data points to $k < d$ dimensions we will get a bad representation. Similarly to supervised learning, we can apply a mapping $\phi(\mathbf{x}_i) = \tilde{\mathbf{x}}_i$ with $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ that will increase the initial dimension of the data to $d' > d$ such that when we apply PCA to project the data to a lower dimension $k < d'$ we get a linearly separable representation. Recall that to avoid the feature explosion we never actually apply the function ϕ, but instead we use the kernel trick.

Lemma 8.3.1 (Kernel PCA Objective) *Let $\tilde{\mathbf{X}} \in \mathbb{R}^{d'\times n}$ be the transformed feature matrix (with $\tilde{\mathbf{x}}_i = \phi(\mathbf{x}_i)$), then the following optimization problems*

are equivalent.

$$\mathbf{w}_j^* = \underset{\substack{\mathbf{w}_j^T\mathbf{w}_j=1 \\ \mathbf{w}_j^T\mathbf{w}_l=0, \forall 1\leq l<j}}{\arg\max} \quad \mathbf{w}_j^T\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\mathbf{w}_j \tag{8.47}$$

$$\boldsymbol{\alpha}_j^* = \underset{\substack{\boldsymbol{\alpha}_j^T\mathbf{K}\boldsymbol{\alpha}_j=1 \\ \boldsymbol{\alpha}_j^T\mathbf{K}\boldsymbol{\alpha}_l=0, \forall 1\leq l<j}}{\arg\max} \quad \boldsymbol{\alpha}_j\mathbf{K}^T\mathbf{K}\boldsymbol{\alpha}_j \tag{8.48}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^n$ and $\mathbf{K} \in \mathbb{R}^{n\times n}$ is the kernel matrix[a] of the features in $\mathbf{X} \in \mathbb{R}^{d\times n}$.

[a] Recall $K_{i,j} := \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j)$

Proof. Assume that any principal component can be written as a linear combination of the form $\mathbf{w}_j = \sum_{k=1}^n \alpha_k^{(j)}\tilde{\mathbf{x}}_k$ where $j \in \{1, \ldots, k\}$ for some $\boldsymbol{\alpha}_j := (\alpha_1^{(j)}, \ldots, \alpha_n^{(j)}) \in \mathbb{R}^n$, then:

Objective

$$\begin{aligned}
\mathbf{w}_j^T\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\mathbf{w}_j &= \sum_{i=1}^n (\mathbf{w}_j^T\tilde{\mathbf{x}}_i)^2 && (8.49)\\
&= \sum_{i=1}^n \left(\left(\sum_{k=1}^n \alpha_k^{(j)}\tilde{\mathbf{x}}_k\right)^T \tilde{\mathbf{x}}_i\right)^2 && (8.50)\\
&= \sum_{i=1}^n \left(\sum_{k=1}^n \alpha_k^{(j)} \left(\tilde{\mathbf{x}}_k^T\tilde{\mathbf{x}}_i\right)\right)^2 && (8.51)\\
&= \sum_{i=1}^n \left(\sum_{k=1}^n \alpha_k^{(j)} \mathrm{k}(\tilde{\mathbf{x}}_k, \tilde{\mathbf{x}}_i)\right)^2 && (8.52)\\
&= \sum_{i=1}^n \left(\boldsymbol{\alpha}_j^T\mathbf{K}_i\right)^2 && (8.53)\\
&= \boldsymbol{\alpha}_j\mathbf{K}^T\mathbf{K}\boldsymbol{\alpha}_j && (8.54)
\end{aligned}$$

Constraints

$$\begin{aligned}
\mathbf{w}_j^T\mathbf{w}_l &= \left(\sum_{k_1=1}^n \alpha_{k_1}^{(j)}\tilde{\mathbf{x}}_{k_1}\right)^T \left(\sum_{k_2=1}^n \alpha_{k_2}^{(l)}\tilde{\mathbf{x}}_{k_2}\right) && (8.55)\\
&= \sum_{k_1=1}^n \sum_{k_2=1}^n \alpha_{k_1}^{(j)}\alpha_{k_2}^{(l)}\tilde{\mathbf{x}}_{k_1}^T\tilde{\mathbf{x}}_{k_2} && (8.56)\\
&= \sum_{k_1=1}^n \sum_{k_2=1}^n \alpha_{k_1}^{(j)}\alpha_{k_2}^{(l)}\mathrm{k}(\mathbf{x}_{k_1}, \mathbf{x}_{k_2}) && (8.57)\\
&= \boldsymbol{\alpha}_j^T\boldsymbol{K}\boldsymbol{\alpha}_l && (8.58)
\end{aligned}$$

□

Lemma 8.3.2 (Kernel PCA) *Let $\mathbf{K} \in \mathbb{R}^{n\times n}$ be the kernel matrix of the features in $\mathbf{X} \in \mathbb{R}^{d\times n}$, $j \in \{1, \ldots, k\}$, and $\mathbf{v}_j$ be the eigenvector associated to*

the j-th largest eigenvalue λ_j of $\mathbf{K}$, then $\boldsymbol{\alpha}_j^* = \frac{1}{\sqrt{\lambda_j}}\mathbf{v}_j$ with

$$\boldsymbol{\alpha}_j^* = \underset{\substack{\boldsymbol{\alpha}_j^T \mathbf{K} \boldsymbol{\alpha}_j = 1 \\ \boldsymbol{\alpha}_j^T \mathbf{K} \boldsymbol{\alpha}_l = 0, \forall 1 \leq l < j}}{\arg\max} \boldsymbol{\alpha}_j \mathbf{K}^T \mathbf{K} \boldsymbol{\alpha}_j \tag{8.59}$$

Computing Kernel PCA To compute the kernelized PCA of some (centered) feature matrix $\mathbf{X} \in \mathbb{R}^{d \times n}$ we apply the following steps:

1. Pick any kernel, even possibly infinite dimensional kernels (*e.g.* a Gaussian kernel $\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(\frac{-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{h^2}\right)$ for $h \in \mathbb{R}$).
2. Compute the kernel matrix using the selected kernel $K_{i,j} = \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j)$.
3. Apply kernel PCA by computing the eigendecomposition $\mathbf{K} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^T$, then $\boldsymbol{\alpha}_j = \frac{1}{\sqrt{\lambda_j}}\mathbf{v}_j$ where $\mathbf{v}_j$ is the j-th column of $\mathbf{V}$ and λ_j is the j-th diagonal of $\boldsymbol{\Lambda}$.
4. Finally pick some output dimension k and to find the reduced vector $\mathbf{z} = (z^{(1)}, \ldots, z^{(k)}) \in \mathbb{R}^k$ (principal scores) of some $\mathbf{x}$ we compute $z^{(i)} = \sum_{j=1}^n \alpha_j^{(i)} \mathrm{k}(\mathbf{x}_j, \mathbf{x})$.

Notes

1. *Kernel K-Means*: if we want to cluster some d dimensional features that are not linearly separable we can apply kernel PCA on the features (*e.g.* with some infinite dimensional kernel) with $k = d$. By having $k = d$ and and infinite kernel we project the initial data to an infinite dimensional space, and then back to the initial dimension d with kernel PCA.
2. *Centering Kernel*: even if our initial features in $\mathbf{X}$ are centered we may get a non-centered kernel matrix $\mathbf{K}$. To solve this problem is good practice to center recenter it as: $\mathbf{K}' = \mathbf{K} - \mathbf{KE} - \mathbf{EK} + \mathbf{EKE}$ where $E_{i,j} = \frac{1}{n}$, $\mathbf{E} \in \mathbb{R}^{n \times n}$.
3. *Uses*: kernel PCA is a very useful method to discover non-linear features before applying any model, included supervised methods (SVM, nerual networks, ...).

8.4 Autoencoders

Autoencoders are an application of neural networks to unsupervised dimension reduction. The key idea works as follows: build a multi-layer neural network f such that the input and output dimension d are the same, then feed some feature vector as input, and train the network to reconstruct the input vector as output, that is find f such that $f(\mathbf{x}, \boldsymbol{\theta}) = \hat{\mathbf{y}} \approx \mathbf{x}$. How does this network reduce the dimension of $\mathbf{x}$? The idea is that we will build the network such that it has some hidden layer $\mathbf{h}^{(l)} \in \mathbb{R}^k$ where $k < d$ (*i.e.* some bottleneck). Then, after the network is trained with back-propagation to reconstruct the input as accurately as possible, if we feed some vector $\mathbf{x}$, the vector $\mathbf{z} := \mathbf{h}^{(\beta)}$ must contain a dimensionally reduced representation of $\mathbf{x}$.

Mathematical View

Goal

Given a set of feature vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ where $\mathbf{x}_i \in \mathbb{R}^d$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$), and the desired output dimension $k \in \mathbb{N}$ with $1 \leq k \leq d$

Output the analog set of dimensionally reduced vectors $\mathbf{z}_1, \ldots, \mathbf{z}_n$ where $\mathbf{z}_i \in \mathbb{R}^k$ (which can be represented as a matrix $\mathbf{Z} \in \mathbb{R}^{n \times k}$) and the parameters $\boldsymbol{\theta}$ such that

$$f(\mathbf{x}; \boldsymbol{\theta}) = \underbrace{f^{(L)}(\cdots \overbrace{f^{(\beta)}(\cdots f^{(1)}(\mathbf{x}; \boldsymbol{\theta}) \cdots ; \boldsymbol{\theta})}^{Encoder} \cdots \boldsymbol{\theta})}_{Decoder} \tag{8.60}$$

satisfies the following optimization for some loss function $\updownarrow_\star$:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \mathbb{R}^D}{\arg\min} \frac{1}{nd} \sum_{i=1}^{n} \sum_{j=1}^{d} \updownarrow_\star (x_{i,j}, f(\boldsymbol{\theta}; \mathbf{x}_i)_j) \tag{8.61}$$

More precisely:

$$\mathbf{h}^{(1)} := f^{(1)}(\mathbf{x}; \mathbf{W}^{(1)}, \mathbf{b}^{(1)}) = \varphi^{(1)}\left(\mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right) \tag{8.62}$$

$$\mathbf{h}^{(2)} := f^{(2)}(\mathbf{h}_1; \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) = \varphi^{(2)}\left(\mathbf{h}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}\right) \tag{8.63}$$

$$\vdots$$

$$\mathbf{h}^{(\beta)} := f^{(\beta)}(\mathbf{h}^{(\beta-1)}; \mathbf{W}^{(\beta)}, \mathbf{b}^{(\beta)}) = \varphi^{(\beta)}\left(\mathbf{h}^{(\beta-1)}\mathbf{W}^{(\beta)} + \mathbf{b}^{(\beta)}\right) \tag{8.64}$$

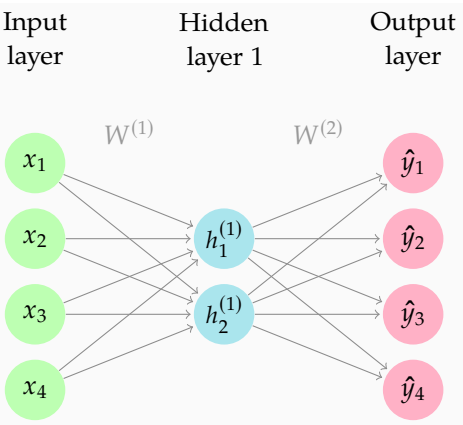
$$\vdots$$

$$\mathbf{h}^{(L)} := f^{(L)}(\mathbf{h}^{(L-1)}; \mathbf{W}^{(L)}, \mathbf{b}^{(L)}) = \varphi^{(L)}\left(\mathbf{h}^{(L-1)}\mathbf{W}^{(L)} + \mathbf{b}^{(L)}\right) \tag{8.65}$$

where the layer β is the bottleneck and $\mathbf{z} := \mathbf{h}^{(\beta)} \in \mathbb{R}^k$.

Graph View

Example 8.4.1 (Autoencoder Graph View) In this example we will have $d = 4$ and $k = 2$, this architecture is as simple as possible, *i.e.* only one encoding and one encoding and decoding function with no biases. The autoencoder neural network graph view is then:



Where $\hat{\mathbf{y}}$ should be as close as possible to $\mathbf{x}$ after training and if we feed $\mathbf{x}$ will have $\mathbf{z} = \mathbf{h}^{(1)}$ as the dimension reduced representation.

Notes

- *Autoencoders PCA*: if we pick the identity function as activation function $\varphi^{(l)}$ for all layers, the autoencoder will have the exact same result as PCA. If, instead, we use non-linear functions $\varphi^{(l)}$ the autoencoder will usually find a better compression than PCA for the same k. The downside is that the optimization is non-convex and thus it relies heavily on the initialization of the weights and biases.
- *Denoising Autoencoders*: a very interesting application of autoencoders is denoising. Denoising is a procedure in which we add a noise vector $\mathbf{n}$ to each input $\mathbf{x}' := \mathbf{x} + \mathbf{n}$ and then train the autoencoder to reconstruct the original $\mathbf{x}$. Since the bottleneck is forced to store only important features of $\mathbf{x}'$ noise will be removed in favor of more important characteristics. Denoising has many applications, one of which is image processing.

Probabilistic Approach to Unsupervised Learning

9

9.1 Mixture Distribution

To understand mixture models consider the following experiment: we are given a set of vectors $\mathbf{x}_1, \dots, \mathbf{x}_N$ with $\mathbf{x}_n \in \mathbb{R}^D$, and we are told that those vectors were drawn from K distinct distributions with probability densities $f_1, \dots, f_K$ with $f_k : \mathbb{R}^D \to [0, 1]$. Let $\mathbf{X}$ be the random vector and $\mathbf{x}_n$ for $n \in \{1, \dots, N\}$ be realizations of $\mathbf{X}$ (where $\mathbf{X} = [\mathsf{X}_1 \ \cdots \ \mathsf{X}_D]^T$, with X_d for $d \in \{1, \dots, D\}$ are random variables). The goal will be to find the probability density function $f_{\mathbf{X}}$ of $\mathbf{X}$, clearly it cannot be a single f_k since each one of them specifies only one of those distributions, thus it must be something more complex. If we knew for each $\mathbf{x}_n$ which of the K distribution is $\mathbf{X}$ choosing for that particular realization, then we could just pick the right f_k, but we have no idea which of the K distributions is chosen each time.

To solve this problem we introduce a new random variable Z that will take the value k when $\mathbf{X}$ is drawing from the k-th distribution. In other words we assume that Z is telling us from which distribution is $\mathbf{X}$ choosing. Clearly we don't know how $\mathbf{X}$ chooses the different distributions and thus we don't know the probability density function $f_{\mathsf{Z}}(k) = \mathbb{P}[\mathsf{Z} = k] : \{1, \dots, K\} \to [0, 1]$ of Z, but we assume that we do and see where this takes us[1] . To reformulate what we have seen before in terms of Z, $\mathbb{P}[\mathbf{X} = \mathbf{x} \mid \mathsf{Z} = k] = f_{\mathbf{X}|\mathsf{Z}}(\mathbf{x} \mid k, \boldsymbol{\theta}_k) = f_k(\mathbf{x} \mid \boldsymbol{\theta}_k)$ (where $\boldsymbol{\theta}_k$ are the parameters of the distribution f_k), *i.e.* if we know that the k-th distribution has been chosen then we know that $\mathbf{X}$ is distributed as f_k.

1: Random variables that are not observed, like Z, are usually called *latent* variables.

Using the previous assumption we can now evaluate the marginal distribution of $\mathbf{X}$ and thus find its probability density function:

$$\mathbb{P}[\mathbf{X} = \mathbf{x}] \coloneqq \sum_{k=1}^{K} \mathbb{P}[\mathsf{Z} = k]\mathbb{P}[\mathbf{X} = \mathbf{x} \mid \mathsf{Z} = k] \tag{9.1}$$

$$= \sum_{k=1}^{K} f_{\mathsf{Z}}(k) f_k(\mathbf{x} \mid \boldsymbol{\theta}_k) \tag{9.2}$$

where the distribution of $\mathbf{X}$ is parametrized by $\boldsymbol{\theta}$ *i.e.* $f_{\mathbf{X}}(\mathbf{x} \mid \boldsymbol{\theta}) = \mathbb{P}[\mathbf{X} = \mathbf{x}]$ with $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_K\}$. Recall that $\mathbb{P}[\mathsf{Z} = k]$ is unknown and thus we can take it one step further and let $\pi_k \coloneqq \mathbb{P}[\mathsf{Z} = k]$ also be parameters of the probability density function of $\mathbf{X}$, *i.e.* $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_K, \pi_1, \dots, \pi_K\}$ hence removing the dependency on Z. Note that since π_k represent probabilities $\sum_{k=1}^{K} \pi_k = 1$.

Definition 9.1.1 (Mixture Distribution) *Let $f_k(\mathbf{x} \mid \boldsymbol{\theta}_k) : \mathbb{R}^D \to [0, 1]$ for $k \in \{1, \dots, K\}$ be K probability density functions of different distributions,*

then we define their mixture distribution *as:*

$$f_{\mathbf{X}}(\mathbf{x} \mid \boldsymbol{\theta}) := \sum_{k=1}^{K} \pi_k f_k(\mathbf{x} \mid \boldsymbol{\theta}_k) \tag{9.3}$$

[a] *with* $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_K, \pi_1, \dots, \pi_K\}$ *as the parameters of* $f_{\mathbf{X}}$. *Where* π_k *are called* mixture weights *and* f_k *are called* mixture components.

[a] We have that $\mathbf{X} \sim f_{\mathbf{X}}(\boldsymbol{\theta})$

9.2 Gaussian Mixtures Model

A Gaussian mixture model is a model used for unsupervised clustering, to understand how this works lets go back to the previous notion of mixture distribution. We will make the assumption that the distributions from which $\mathbf{x}_1, \dots, \mathbf{x}_N$ are drawn are K multivariate Gaussians, *i.e.* $f_k(\mathbf{x} \mid \boldsymbol{\theta}_k) := \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ for $k \in \{1, \dots, K\}$. In this case we have decided to use the same distribution with different parameters for all of the K mixture components, but different models exist that use different assumptions. This will give the following mixture distribution for $\mathbf{X}$:

$$f_{\mathbf{X}}(\mathbf{x} \mid \boldsymbol{\theta}) := \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{9.4}$$

with $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_1, \dots, \boldsymbol{\Sigma}_K\}$. It's important to realize that if we had $\boldsymbol{\theta}$, then we could also compute $\gamma_k(\mathbf{x}) := \mathbb{P}[\mathsf{Z} = k \mid \mathbf{X} = \mathbf{x}]$ *i.e.* given any new $\mathbf{x}$ what is the probability that it's part of the mixture k. This function $\gamma_k(\mathbf{x})$ is crucial for clustering since it tells us which cluster (mixture) is $\mathbf{x}$ more probable to be part of. Note that we can evaluate $\gamma_k(\mathbf{x})$ by using Bayes Theorem as follows:

$$\gamma_k(\mathbf{x}) := \mathbb{P}[\mathsf{Z} = k \mid \mathbf{X} = \mathbf{x}] = \frac{\mathbb{P}[\mathsf{Z} = k]\mathbb{P}[\mathbf{X} = \mathbf{x} \mid \mathsf{Z} = k]}{\sum_{k'=1}^{K} \mathbb{P}[\mathsf{Z} = k']\mathbb{P}[\mathbf{X} = \mathbf{x} \mid \mathsf{Z} = k']} \tag{9.5}$$

$$= \frac{\pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'=1}^{K} \pi_{k'} \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})} \tag{9.6}$$

We can now write the concrete goal of a Gaussian Mixture model.

Goal

Given a set of feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ where $\mathbf{x}_n \in \mathbb{R}^D$ (which can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$), and a desired number of clusters $K \in \mathbb{N}$.

Output the parameters $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_1, \dots, \boldsymbol{\Sigma}_K\}$ with $\boldsymbol{\pi} \in \mathbb{R}^K$, $\boldsymbol{\mu}_k \in \mathbb{R}^D$, $\boldsymbol{\Sigma}_k \in \mathbb{R}^{D \times D}$ that characterize

$$\gamma_k(\mathbf{x}) := \frac{\pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'=1}^{K} \pi_{k'} \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})} \tag{9.7}$$

Now that we have clearly defined our goal only one part is missing: the estimation of the parameters $\boldsymbol{\theta}$ given the concrete realizations $\mathbf{x}_1, \dots, \mathbf{x}_N$.

To do so we will evaluate the maximum likelihood estimation (MLE) of $f_{\mathbf{X}}(\mathbf{x} \mid \boldsymbol{\theta}) = \mathbb{P}[\mathbf{X} = \mathbf{x}]$ which gives us the following log likelihood function:

$$\ln \mathbb{P}[\mathbf{X}_1 = \mathbf{x}_1, \ldots, \mathbf{X}_n = \mathbf{x}_n] \overset{i.i.d.}{=} \ln \prod_{n=1}^{N} \mathbb{P}[\mathbf{X}_n = \mathbf{x}_n] \tag{9.8}$$

$$= \sum_{n=1}^{N} \ln \mathbb{P}[\mathbf{X}_n = \mathbf{x}_n] \tag{9.9}$$

$$= \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{9.10}$$

$$=: \mathrm{LL}(\boldsymbol{\theta}) \tag{9.11}$$

Then since $\boldsymbol{\theta} = \{\pi, \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_K\}$ we have to differentiate LL with respect to $\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$ which gives:

$$\frac{\partial}{\partial \pi_k} \left[\mathrm{LL}(\boldsymbol{\theta}) + \lambda \left(\sum_{k=1}^{K} \pi_k - 1 \right) \right] \overset{!}{=} 0 \Rightarrow \pi_k = \frac{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)}{N} \tag{9.12}$$

$$\frac{\partial}{\partial \boldsymbol{\mu}_k} \left[\mathrm{LL}(\boldsymbol{\theta}) \right] \overset{!}{=} 0 \Rightarrow \boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n) \mathbf{x}_n}{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)} \tag{9.13}$$

$$\frac{\partial}{\partial \boldsymbol{\Sigma}_k} \left[\mathrm{LL}(\boldsymbol{\theta}) \right] \overset{!}{=} 0 \Rightarrow \boldsymbol{\Sigma}_k = \frac{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T}{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)} \tag{9.14}$$

Where on the first parameter we use the Lagrangian to enforce the constraint that $\sum_{k=1}^{K} \pi_k = 1$. Usually at this point we would solve the system of equations to obtain $\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$, to find the maximum of the log likelihood. However, in this case the equations are coupled and hard to solve jointly we thus use an iterative algorithm called *Soft EM* that works as follows:

Algorithm 9.1: Soft EM Algorithm

$\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k$ ▷ *Initialize means, covariances, and mixing coefficients for all* $k \in \{1, \ldots, K\}$.
while not converged
  for each $n \in \{1, \ldots, N\}$
    $\gamma_k(\mathbf{x}_n) := \frac{\pi_k \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'=1}^{K} \pi_{k'} \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})}$ ▷ *E Step.*
    $\pi_k = \frac{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)}{N}$ ▷ *M Steps*
    $\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n) \mathbf{x}_n}{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)}$
    $\boldsymbol{\Sigma}_k = \frac{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T}{\sum_{n=1}^{N} \gamma_k(\mathbf{x}_n)}$
end
return $\boldsymbol{\theta} = \{\pi, \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_K\}$

Finally, we have obtained an approximation $\hat{\boldsymbol{\theta}}$ which can be used to perform clustering. Note that the Soft EM algorithm usually converges to different local minima depending on the initialization step.