Machine learning for physicists [PHYS-467]

Lecture Notes

Prof. Lenka Zdeborová
TAs: Federica Gerace, Giovanni Piccioli, Alessandro Favero

First edition by the students of the 2021 course on behalf of Prof. Lenka Zdeborová.

Institute of Physics – IPHYS
École polytechnique fédérale de Lausanne – EPFL
January 2022

Contents

1 Lecture 1: ML & Physics, Linear Regression basics 24.9.

Students designated to write this section: Wyler, Verdier, Torre, Toffenetti, Tibke, Tanner
Coordinator: Torre
TA: Giovanni (TA approved)
Students who actually contributed: Tibke, Toffenetti, Torre, Verdier

1.1 Introduction

The immense availability of data and its access are one of the most distinctive features of our age and modern science, including physics, where data analysis has spread to any sub-area, becoming one of the crucial aspects for research success.
This "big data" revolution has been triggered by an exponential growth in memory storage and computing power, commonly known as Moore's law. As a result, computations that were unthinkable just a few decades ago can now be easily performed on a laptop. At the same time, current high-tech industries are promoting the development of specialized computing machines, continuing this trend towards cheap, large-scale computation, suggesting that the "big data" revolution will probably be part of our future as well.

Moreover, this increase in computational capacity has been coupled with new methods for analyzing and learning from large datasets. Typically, these techniques and algorithms stem from physics, statistics, computer science, and computational neuroscience ideas.

Therefore, Machine Learning (ML) and data science are playing increasingly important roles in many aspects of modern technology, ranging from biotechnology to smart engineering devices, from astrophysics to typical daily applications. Consequently, having a thorough understanding of the concepts and tools used in ML is becoming a crucial skill in all sciences and engineering areas.

The purpose of this course, therefore, is to introduce the core concepts and tools of machine learning to physicists. The course begins by covering fundamental concepts in ML and modern statistics, before moving on to more advanced topics in both supervised and unsupervised learning. Topics covered in the course include Linear Regression, Statistical inference and Statistical physics, Supervised and Unsupervised learning, Neural Networks and Deep Neural Networks.

Furthermore, throughout the lectures, we will emphasize the many natural connections between ML and statistical physics. Indeed, many concepts build upon the considerable knowledge most physicists already possess in statistical physics to introduce most of the central ideas and techniques used in modern ML. Physicists are uniquely situated to benefit from and contribute to ML: Plenty of the core concepts and techniques used in ML have their origins in physics. For instance, Monte-Carlo methods, simulated annealing [1] and variational methods are core techniques of ML, while "energy-based models" inspired by statistical physics are the backbone of many deep learning methods.

What is Machine Learning?

Machine Learning is a subset of Artificial Intelligence (AI) that focuses on the study of algorithms that learn to make predictions from a dataset without being explicitly programmed to do so. For instance, an artificially intelligent agent needs to be able to recognize objects in its surroundings and predict the behaviour of its environment in order to make informed choices.

Just to understand the potential of ML, we can mention a few key steps of its history. In the end of the 90's, Deep Blue, a chess-playing computer developed by IBM, was the first computer to win both a chess game and a chess match against the world champion, Garry Kasparov, under regular time controls [2].

At the time Deep Blue was perceived as the most advanced state a computer could reach and beating a Go world champion seemed absolutely infeasible. Go is considered, under an algorithmic point of view, much more difficult than other games such as chess: Its much larger branching factor makes it extremely complicated to use traditional AI methods such as alpha–beta pruning, tree traversal and heuristic search [3]. However, in 2016 AlphaGo succeeded in winning against the world champion, Lee Sedol, in a 5 round game [4–7]. AlphaGo and its successors used a Monte Carlo tree search (MCTS) algorithm to find its moves based on knowledge previously acquired by Machine Learning. Specifically, this is done by an artificial neural network – a deep learning approach – with extensive training, both from human and computer play. A neural network is trained to identify the best moves and the probability of winning of these moves. This neural network, a standard example of reinforcement learning, improves the strength of the tree search, resulting in better move selection in the next step.

Moving more into the biological context, the same company that developed Alpha Go, Deep Mind, also demonstrated how Artificial Intelligence research can drive and accelerate new scientific discoveries. In 2018, [8, 9], and 2020, [10], Deep Mind has succeeded – at least partially – in solving one of the biggest unsolved problems in biology: The prediction of the 3D structure of a protein based solely on its amino acids sequence. They beat every competitor in the CASP (Critical Assessment of protein Structure Prediction) challenge, a worldwide experimental challenge for protein structure prediction, taking place every two years since 1994.

Therefore, nowadays there are big hopes in ML and in AI in general, in particular for solving the most crucial problems of our era, like climate change or personalized medicine. It is clear that ML is changing science in its core. The advancement of Artificial Intelligence will probably mark the 4^{th} industrial revolution, also impacting our daily life and society.

Why a special course for physicists?

Machine learning courses for computer scientist or engineers often focus on their application. The mathematical principles underlying AI, but also of Machine Learning, however, is still not rigorously explained. For many successful applications a through mathematical explanation is still missing. Machine Learning and AI in general needs to be advanced and understood better in order to unleash its full potential in science. When using machine learning, a physicist should be required not only to apply learned techniques, but also to understand them. This course has the aim to provide a tool to do so. Furthermore, physicists have helped developing key concepts in Machine Learning. This lecture will establish links with statistical mechanics such as Gibbs sampling and the Boltzmann machine and therefore allow the understanding of basic Machine Learning concepts.

Examples

We will start with several examples of Machine Learning applications to give a broad idea of different approaches.

A) Classification of images.

Let us suppose that we have a dataset of pictures of cats and dogs with different labels (i.e. $+1$ for a cat and -1 for a dog). What we want from an algorithm is to learn a function that, given a picture, returns the correct label corresponding to dog or cat, thus:

$$f(\text{image}) = \begin{cases} +1 & \text{Cat} \\ -1 & \text{Dog} \end{cases}$$

such that $f(\text{image})$ works on samples unknown by the algorithm. Therefore, we should have a **training set** – a subset used to train the model, and a **test set** – a subset used to test or evaluate the trained model.
Note that nowadays basic classification of images is easy to do, but 20 years ago it was a difficult task.

B) Prediction of survival time of diagnosed cancer patients after treatment.

In this more helpful and interesting example an algorithm could inform whether to use or not a specific treatment on a specific patient. In this case, the training set could be composed of many previous patient information related to their detailed medical and personal history. Then we should find a function that tells us the time that the patient will survive based on the data about the patient. Thus, we should find a function like:

$$f(\text{Data about patients}) = \text{Amount of time they will survive}$$

C) Writing novels.

Can a Machine Learning algorithm write a novel based on a database of existing novels? The difficulty of a task like this is that the mathematical model needs to understand the semantic of a text. In order to write a good novel, it is obviously not sufficient to just concatenate words which may make sense together. A good novel speaks to the reader through messages between the lines. The algorithm must acquire this capability through the given training sets. In fact, a complicated task like this is so far out of reach of the current methods.

As a simpler goal, well within the reach of current methods, which is used on an everyday basis by many people are language translation platforms such as Google Translate or *deepl* [11].

D) Clustering of data.

Design an algorithm finding groups of data points that have some key elements in common. For example, by knowing the "friendship" network of Facebook, is it possible to guess which people were together in a high school class?

E) Play and win a Go match.
Could an AI algorithm beat the Go world champion? We have already seen that it can. Here, the interactions with an opponent and continuous adaption of the own "strategies" is key.

Looking at these examples we can define the main categories and approaches that represent three areas of machine learning. They are briefly summarized in the table below:

Category/Task	Description	Methods	Examples
Supervised learning	Training set of labelled data available to predict labels for unseen data.	$y \in \mathbb{R}$ Regression	B
		$y \in \mathbb{Z}$ Classification	A
Unsupervised learning	No labels available. Find patterns underlying the training data.	Dimensionality Reduction Representation Learning	D
	Generate data from the same distribution as training set.	Generative models	C
Reinforcement learning	Agent takes actions in an environment in order to maximize overall reward.		E

In this course we will focus on supervised and unsupervised learning.[1] The literature on reinforcement learning is extensive and uses ideas and concepts that, to a large degree, are distinct from supervised and unsupervised learning tasks.

1.2 Linear Regression

1.2.1 A simple example of Linear Regression

Consider a train driving with constant velocity in y-direction. Its movement in time[2] x is given by the function

$$y(x) = w_1 x + w_2, \tag{1.1}$$

which describes a line with slope w_1 and bias w_2. Suppose we measure n data pairs $((x_i, y_i))_{i=1}^n$ for the position y_i of the train at time x_i. We distinguish between two goals, which are closely related but not identical:

A) Estimation: Finding the optimal parameters w_1, w_2 for the model (1.1) to describe the available data $((x_i, y_i))_{i=1}^n$.

B) Prediction: Use the available data to make predictions about previously unseen data. This correspond to the question: Given a new time x_{new}, at which position y_{new} is the train?

In general, relationship (1.1) will not hold precisely, because the measured data points are subject to errors $(\varepsilon_i)_{i=1}^n$:

$$y_i = w_1 x_i + w_2 + \varepsilon_i. \tag{1.2}$$

This relationship between the true (but unobserved) underlying parameters w_1 and w_2 and the data points is called a linear regression model. The goal is to find estimated values $\hat{w}_1, \hat{w}_2$ for the parameters w_1, w_2 which would provide the best fit in some sense to the available data. Here, it is common to use the *ordinary least squares* (OLS) method. The reason for choosing this method, as well as alternative methods, will be discussed in Lecture 3. The OLS method minimizes the so-called least square loss function $\mathscr{L} \equiv \mathscr{L}_{OLS}$, which is defined as the sum of squared errors ε_i:

$$\mathscr{L}(w) = \frac{1}{n}\sum_{i=1}^{n} \varepsilon_i^2 = \frac{1}{n}\sum_{i=1}^{n} (y_i - w_1 x_i - w_2)^2. \tag{1.3}$$

It quantifies how costly the mistakes on the prediction are. The necessary conditions that $\mathscr{L}$ has a minimum are:

$$\begin{aligned} 0 &\overset{!}{=} \frac{\partial \mathscr{L}(w)}{\partial w_1} = -2\frac{1}{n}\sum_{i=1}^{n} (y_i - w_1 x_i - w_2)\, x_i \\ 0 &\overset{!}{=} \frac{\partial \mathscr{L}(w)}{\partial w_2} = -2\frac{1}{n}\sum_{i=1}^{n} (y_i - w_1 x_i - w_2)\,. \end{aligned} \tag{1.4}$$

[1] Reinforcement learning will not be covered in this course. Look at the spring semester course CS-456 by Prof. Gerstner.

[2] We are aware that normally the variable t would be chosen to denote the time. However, for the benefit of consistency in what follows we chose x.

This system of equations can be solved by multiplying the first equation by n and the second equation by $\sum_{i=1}^{n} x_i$ and then subtracting the two.[3] As a result, we find the estimated values $\hat{w}_1$ and $\hat{w}_2$ which minimize $\mathscr{L}$:

$$\hat{w}_1 = \frac{\overline{xy} - \overline{x} \cdot \overline{y}}{\overline{x^2} - \overline{x}^2}, \quad \hat{w}_2 = \overline{y} - \overline{x} \cdot \frac{\overline{xy} - \overline{x} \cdot \overline{y}}{\overline{x^2} - \overline{x}^2}. \tag{1.5}$$

Note that for a generic variable $x = (x_1, \ldots, x_n)$, we defined its mean as:

$$\overline{x} \coloneqq \frac{1}{n} \sum_{i=1}^{n} x_i.$$

The simple linear regression problem we just introduced can be rewritten into matrix notation as follows.

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix}.$$

This notation will be used further on to describe more complex situations.

1.2.2 Matrix Notation

We now introduce the matrix notation which will be used during most of the lectures.

X	input data	$\in \mathbb{R}^{n \times d}$	matrix
y	output labels	$\in \mathbb{R}^{n}$	vector
w	weights to learn	$\in \mathbb{R}^{d}$	vector

Where n is the number of samples and d the dimension of the input samples. In the example from before we have:

$$y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}, \quad w = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}.$$

It will be helpful for the discussion of linear regression to define one last piece of notation. For any real number $p \geq 1$, we define the L^p norm of a vector $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ as follows [12]:

$$||\boldsymbol{x}||_p = \left(|x_1|^p + \ldots + |x_d|^p\right)^{\frac{1}{p}}. \tag{1.6}$$

Using the matrix notation, we can express the loss function as:[4]

$$\mathscr{L}(w) = \frac{1}{n} \sum_{i=1}^{n} (y_i - (Xw)_i)^2 = \frac{1}{n} ||y - Xw||_2^2. \tag{1.8}$$

[3] After rewriting (1.4) above we find for the first equation:

$$\sum_i^n x_i y_i = \sum_i^n (w_1 x_i^2) + w_2 x_i)$$
$$n \sum_i^n x_i y_i = n \sum_i^n (w_1 x_i^2) + n w_2 \sum_i^n x_i$$

and for the second one:

$$\sum_i^n y_i = \sum_i^n (w_1 x_i) + n w_2$$
$$\sum_i^n (x_i) \sum_i^n (y_i) = w_1 \left(\sum_i^n (x_i) \right)^2 + n w_2 \sum_i^n x_i$$

Now one equation can be subtracted from the other to get rid of w_2 and to find w_1 after rearranging. Finding w_2 is then straight forward.

[4] For completeness note that equation (1.2) can be written in matrix notation as:

$$y = Xw + \epsilon \tag{1.7}$$

where $\epsilon \in \mathbb{R}^n$ is the vector containing the errors between the measured datapoints and the ground truth.

Now we are interested in finding the w that minimizes the loss function, $\hat{w}$. Setting $\frac{\partial \mathscr{L}(w)}{\partial w_k} \stackrel{!}{=} 0$ for $k \in \{1 \dots d\}$ yields:

$$0 = \frac{\partial \mathscr{L}(w)}{\partial w_k} = -2\frac{1}{n}\sum_{i=1}^{n}(y_i - (Xw)_i) \cdot X_{ik} \tag{1.9}$$

from this we can write:

$$(X^T y)_k = \sum_{i=1}^{n} y_i X_{ik} = \sum_{i=1}^{n}(Xw)_i X_{ik} = \sum_{i=1}^{n}\sum_{j=1}^{d} w_j X_{ij} X_{ik} = (X^T X w)_k \tag{1.10}$$

Since this holds for all $k \in \{1 \dots d\}$ we have

$$X^T y = X^T X \hat{w}. \tag{1.11}$$

Now, assuming $X^T X$ invertible, we derive that $\hat{w} = \left(X^T X\right)^{-1} X^T y$. The matrix $X^T X \in \mathbb{R}^{d \times d}$ is called the Covariance matrix and $X^+ \coloneqq (X^T X)^{-1} X^T$ the pseudo-inverse of X.

More precisely, in the definition of the Moore-Penrose pseudo-inverse of a matrix we distinguish mainly two cases. When X has linearly independent columns (and thus matrix $X^T X$ is invertible), the pseudo-inverse X^+ can be computed as

$$X^+ = (X^T X)^{-1} X^T.$$

This particular pseudo-inverse represents a *left inverse* [13].

When X has linearly independent rows (and thus matrix $X X^T$ is invertible), the pseudo inverse X^+ can be computed:

$$X^+ = X^T (X X^T)^{-1}$$

which instead represents a *rigth inverse* [13].

1.2.3 Regularization

The invertibility of $X^T X$ depends on the relationship between the number of samples n and the dimension d of the input samples. Typically, it is possible to distinguish two situations:

- $n > d$: $X^T X$ is typically invertible. Mathematically, this is only true if there are at least d rows that are linear independent. However, for real (usually noisy) data we would typically have $\text{rank}(X^T X) = d$.
- $n < d$: $X^T X$ is never invertible. This is easy to see because:

  $$\text{rank}(X^T X) \leq \min\left(\text{rank}(X^T), \text{rank}(X)\right) = \text{rank}(X) \leq \min(d, n) = n.$$

  Since $X^T X$ is a $d \times d$ matrix with $\text{rank}(X^T X) \leq$ n, $X^T X$ is not invertible. Note that the kernel of $X^T X$ has at least dimension $d - n$.[5]

Formally speaking, if $\text{rank}(X) = d$, namely, the predictors $X_{:,1}, ..., X_{:,d}$ (i.e. columns of X) are linearly independent, then $\hat{w}$, the minimizer of (1.8), is unique. In the case of $\text{rank}(X) < d$, which happens when $d > n$, $X^T X$ is singular, implying there are infinitely many solutions to the least squares problem (1.8).

There is no solution which could be a priori chosen. Therefore, we should ask ourselves which w should we pick in the case in which $d > n$?

The most common choice aims to choose w with the smallest norm $\sum_k w_k^2$. This is called **square regularization** or **$\mathbf{L_2}$ regularization**. The loss function (1.8) can then be reformulated as:

$$\mathscr{L}(w) = \frac{1}{n}\sum_{i}^{n}(y_i - (Xw)_i)^2 + \lambda\frac{1}{n}\sum_{k} w_k^2 \tag{1.12}$$

where λ represents the regularization strength. Again, we minimize for w:

$$0 = n\frac{\partial \mathscr{L}(w)}{\partial w_k} = -2\sum_{i}(y_i - (Xw)_i)X_{ik} + 2\lambda w_k \tag{1.13}$$

[5] I.e. $\{w \in \mathbb{R}^d : X^T X w = 0\}$ is a vector space with at least dimension $d - n$.

Similar as above we then derive $X^T y = X^T X w + \lambda w = \left(X^T X + \lambda \mathbb{I}\right) w$ and find $\hat{w}$ as:

$$\hat{w} = \left(X^T X + \lambda \mathbb{I}\right)^{-1} X^T y \text{ , if } \left(X^T X + \lambda \mathbb{I}\right) \text{ is invertible, which for } \lambda > 0 \text{ is always the case.} \tag{1.14}$$

Now, the question may arise while we discuss the situation $n < d$? If we go back to the example of predicting the survival time of diagnosed cancer patients after treatment, we can easily realize that in this problem n represents the number of patients, d the amount of different information we have for each patient (e.g. many parameters of the case history, anamnesis and treatment, as well as more "meta data" like parameters as age or sex). Therefore, considering that for each patient we might have a lot and even more parameters than the whole number of patients, we could have $n \ll d$. In this case $X^T X$ would not be invertible.

2 Lecture 2: Linear regression continued & Bayesian inference 1.10.

Students designated to write this section: Salvatore Alessandro, Sandgathe Nolan James, Sangwan Nikunj, Sinibaldi Alessandro, Smith Lana Maria, Soutter Jacques Bernard Jean
Coordinator: Sinibaldi Alessandro
TA: Alessandro (TA approved)
Students who actually contributed: Sinibaldi Alessandro, Nikunj Sangwan, Lana Maria Smith, Nolan James Sandgathe, Alessandro Salvatore, Soutter Jacques Bernard Jean

2.1 Polynomial Regression

A special case of linear regression is *polynomial regression*, where the relationship between the data x and the label y is given by a degree-p polynomial. We can define $\mu = 1, \ldots, n$ as the index for sample points and $i = 1, \ldots, d$ as the index for the dimension, so the degree (or grade) p of the considered polynomial is given by $p = d - 1$. Hence, in general, the vector w will belong to the space $\mathbb{R}^{p+1}$, the vector y to $\mathbb{R}^n$ and the matrix X to $\mathbb{R}^{n \times (p+1)}$. Given the degree p polynomial $y = \sum_{k=1}^{p} w_{p+1-k} x^k + w_{p+1}$, we can write:

$$w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_{p+1} \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} x_1^p & x_1^{p-1} & \ldots & x_1 & 1 \\ x_2^p & x_2^{p-1} & \ldots & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^p & x_n^{p-1} & \ldots & x_n & 1 \end{pmatrix}. \tag{2.1}$$

In particular, the simplest case is a 2^{nd} order polynomial $y = ax^2 + bx + c$, for which we have:

$$w = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n^2 & x_n & 1 \end{pmatrix}. \tag{2.2}$$

Using (2.1), we can solve the regularized regression problem by minimizing the loss function

$$\mathscr{L}(w) = \frac{1}{n} \sum_{\mu=1}^{n} \left(y_\mu - \sum_{i=1}^{d} X_{\mu i} w_i \right)^2 + \lambda \frac{1}{n} \sum_{i=1}^{d} w_i^2. \tag{2.3}$$

As we know, the solution of (2.3) is given by the weights $\hat{w} = \left(X^T X + \lambda \mathbb{I}\right)^{-1} X^T y$.
Let us make some considerations on the existence of solutions depending on the relationship between n and p. In doing this, we ignore the regularization term ($\lambda = 0$). Three cases can occur:

- if $p \ll n$, we have that typically the set of equations $y_\mu = \sum_{i=1}^{d} X_{\mu i} w_i \ \forall \mu$ will not have a solution, as the system has more (linearly independent) equations than variables (parameters). In this case, the matrix $X^T X$ is *invertible*;
- if $p + 1 = n$, we have typically a unique solution since there is the same number of variables and equations;
- if $p + 1 \gg n$, we have many solutions, as the system has more variables than equations. In this case, $X^T X$ is *not invertible*.

Notice that in the case $p+1=n$ we have typically a unique polynomial that passes exactly through all the sample points. For $p+1 \gg n$, we have many solutions that fit perfectly all the points. The regularization terms with $\lambda > 0$ chooses among them.

To assess the quality of the chosen solution or how to select the right value for the regularization strength λ the error of the prediction on new data must be analyzed.

2.2 Overfitting and the Bias-Variance Trade-Off

The goal of regression is often trying to correctly predict the labels of new data points. With this in mind, when provided with a single dataset, it should be split into a *training*, a *test* and a *validation dataset*. The training dataset will be used to tune the parameters of the function fitting the training data, while the test dataset will only be "seen" by the algorithm at the end of the process to evaluate its performance. The quantity used to evaluate performance is the test error, which has often the same form as the loss function (1.12), but without the regularization term. Hence, for a regression problem, the test error is defined as:

$$E_{\text{test}} = \frac{1}{n_{\text{test}}} \sum_{i \in \text{test set}}^{n_{\text{test}}} (y_i - X_i \cdot \hat{w})^2, \tag{2.4}$$

with n_{test} the number of data points in the test dataset. The equation (2.4) corresponds to the average quadratic deviation of the predicted values and the labels of the points from the test data.

Using too many parameters, too large p, will lead the algorithm to overfit, reaching zero training error by fitting the noise of the training set as well as the actual underlying process (semantic). Therefore, such an algorithm would typically not be able to correctly predict points which deviate from the training data because it would be too sensitive to the noise present in the training dataset. On the other hand, underfitting – using too few parameters – would impose an excessive bias on the model since it would be restricted to explore a too small subspace of the whole space of polynomials, where we can think that the ground truth function resides.

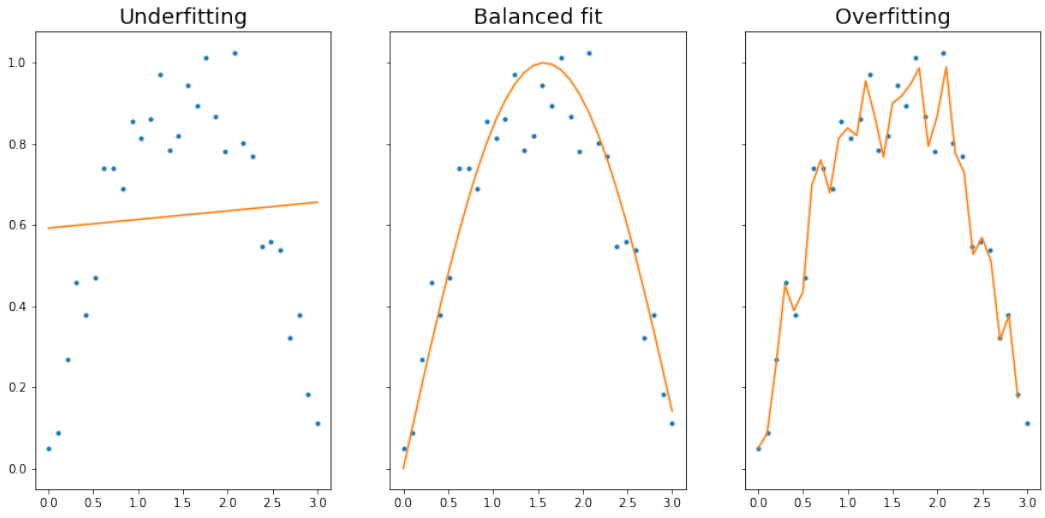


Figure 1: Different types of fit for Polynomial data with noise

It is possible to observe in Figure 1 three different case of fitting for some generated polynomial data with noise. On the left one can see a case of underfitting where high degree polynomial data is fitted with a linear fit, in this case the fit does not have the flexibility to capture the necessary features of the data. On the other hand, on the right a case of overfitting, where the fitting is done with too many parameters and the model fits even the noise in the data which is an unwanted feature is shown.

It is now clear, that the degree of the polynomial we are trying to fit is a crucial parameter (it is more precise to call it a hyperparameter, since it is not tuned during the training by minimizing the validation loss). As we have just

said, we cannot optimize p on the training set (since bigger p would always lead to smaller training errors), nor can we use the test set, which should be only seen by the final regression model. This is the reason why we generated the validation dataset, which is basically a "hybrid" test-train dataset on which we can evaluate the model to tune the hyperparameters, as the degree of the polynomial p.

The validation error as a function of the degree of the polynomial usually follows a U-shaped curve conceptually similar to the one shown in the image below because of what is known as bias-variance trade off, which will be further explored in the following paragraph.[6] Therefore, we can choose the degree of the polynomial among the values around the minimum of the attained plot by using the validation dataset. This procedure can be replicated for every combination of hyperparameters to find the best one.

It is interesting to anticipate that, overparameterizing in addition with techniques to avoid overfitting the data is the key to be able to model complex data and extract implicit features. We will discuss this in Lecture 13.

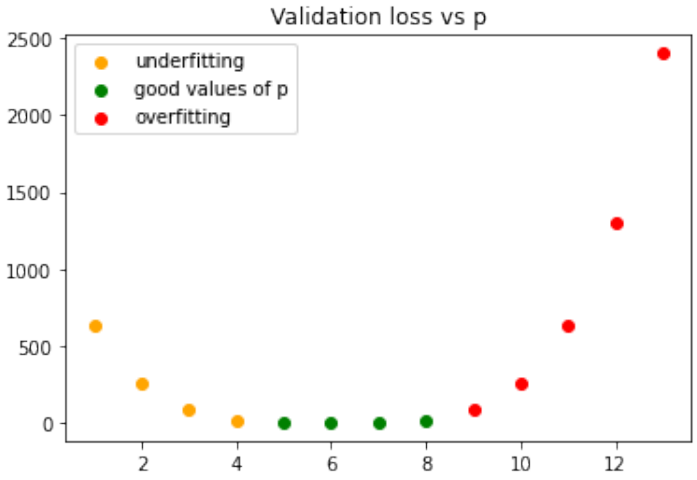


Figure 2: Underfitting and overfitting

The concept of overfitting is often formalized using the so-called *bias-variance* trade-off. Let us call the function generating the labels from the input data the *ground truth* function. Regression aims to make predictions by approximating the ground truth function. Errors originate mainly from two different sources: bias and variance. Errors due to bias are because the class of functions that the considered models represent is far away from the ground truth function. The bias is typically high when the number of parameters is small and close to zero when the number of parameters is large. Errors due to variance are the fluctuations of the fitted model prediction due to noise in the dataset (the finite size of the training set, label noise etc). The variance is typically small when the number of parameters is small and may grow when the number of parameters is large and able to fit the noise. A model can have:

- high bias: The best fitting model is far from the ground truth function.
- low bias: The best fitting models if close to the ground truth function.
- high variance: Prediction changes a lot when small noise is added or the dataset slightly adjusted.
- low variance: Prediction changes little when small noise is added or the dataset slightly adjusted.

In figure 3 the high bias and low variance case corresponds to a situation with underfitting, whereas the high variance and low bias case corresponds to the overfitting situation.

[6]Note that the bias-variance curve of neural networks differ from the one discussed here. In particular after reaching in the overfitting region a local maximum the test error will decrease when adding even more parameters to tune. This will be discussed later.

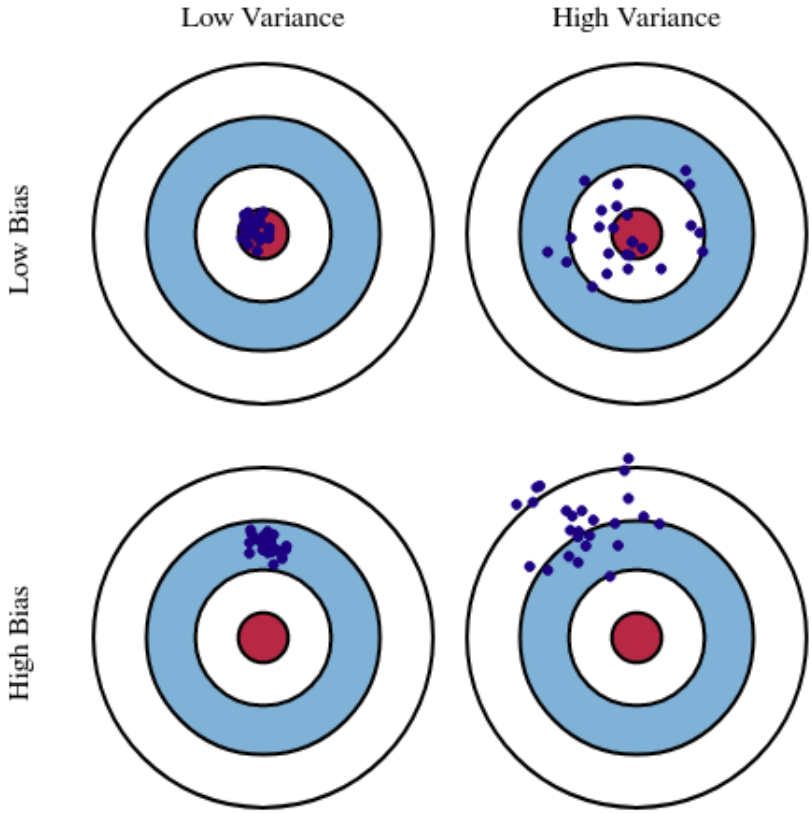


Figure 3: Difference between Bias and Variance. The figures can be interpreted as follows: The target is the ground truth of one component of the weight. Each point denotes an estimation of the the same component where the the measured values (input) for the training vary a little from point to point.

We are looking for low bias and variance at the same time. This corresponds to finding a good balance between underfitting and overfitting. The area with a good balance (green area in Fig. 2) is often called the *sweet spot*.

2.3 Bayesian Inference

In what follows we will provide a short refresher of the basics of probability before considering the Bayes' rule and inference problems.

Basics of probability

1. Random variable X, taking values from alphabet $\mathscr{A}_x = \{a_1, ..., a_I\}$ with probabilities $P_x = \{p_1, ..., p_I\}$ we have:

$$P(x = a_i) = p_i \quad \text{and} \quad \sum_i P(x = a_i) = 1\,. \tag{2.5}$$

2. Joint probability: $P(X, Y)$. For independent random variables X and Y we have:

$$P(X, Y) = P(X) \cdot P(Y)\,. \tag{2.6}$$

3. Marginal probability: Given the joint probability $P(X, Y)$, the marginal probabilities are defined as:

$$\begin{aligned} P(X) &= \sum_{y \in \mathscr{A}_y} P(X, Y)\,, \\ P(Y) &= \sum_{x \in \mathscr{A}_x} P(X, Y)\,. \end{aligned} \tag{2.7}$$

4. Conditional probability: Probability that the random variable X is equal to a_i, given the random variable Y is equal to b_j:

$$P(X = a_i|Y = b_j) = \frac{P(X = a_i, Y = b_j)}{P(Y = b_j)}\,. \tag{2.8}$$

Properties of conditional probabilities

1. Marginal probability of conditional probability. The expressions below can be found by using (2.7) and (2.8).[7]

$$\begin{aligned} \sum_{a_i \in \mathscr{A}_X} P(X = a_i|Y = b_j) = 1\,, \\ \sum_{b_j \in \mathscr{A}_Y} P(Y = b_j|X = a_i) = 1\,. \end{aligned} \tag{2.9}$$

2. Again for two random variables X and Y it holds that:

$$P(X, Y) = P(X|Y)P(Y) = P(Y|X)P(X)\,. \tag{2.10}$$

3. Bayes' rule denotes the following relation derived from (2.10):

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}\,. \tag{2.11}$$

Elementary example of Bayes' Rule

Mike wants to take a COVID-19 test.[8] In this example "a" represents the state of Mike, then $a = 1$ signifies that he is infected and $a = 0$ means that he is not infected. The variable "b" represents the result of the COVID test, then $b = 1$ means that the result was positive and $b = 0$ means that the result was negative. We assume that:

1. The test is 95% reliable, i.e. 95% of the people who are infected test positive. 95% of the people who are healthy test negative.

2. Mike does not have any symptoms and given all we know about him (without the result of the test) only 1% of the same people of the considered population infected.

Question: Mike tests positive. What is the probability that Mike is infected?
Solution: From the information that the test is 95% reliable, we know that:

$$\begin{aligned} P(b = 1|a = 1) = 0.95, \\ P(b = 0|a = 0) = 0.95. \end{aligned} \tag{2.12}$$

Thanks to equation (2.5) we can compute the probability that the test returns a negative result if the patient is actually infected $P(b = 0|a = 1)$ and the probability that the test returns a positive result if the patient is not infected $P(b = 1|a = 0)$:

$$\begin{aligned} P(b = 0|a = 1) = 1 - P(b = 1|a = 1) = 0.05, \\ P(b = 1|a = 0) = 1 - P(b = 0|a = 0) = 0.05. \end{aligned} \tag{2.13}$$

Mathematically, the fact that only 1% of the population is infected can be written as:

$$\begin{aligned} P(a = 1) = 0.01, \\ P(a = 0) = 0.99. \end{aligned} \tag{2.14}$$

[7] In particular and more general for two random variables X and Y:
$\sum_{x \in \mathscr{A}_X} P(X = x|Y = y) = \sum_{x \in \mathscr{A}_X} \frac{P(X=x,Y=y)}{P(Y=y)} = \sum_{x \in \mathscr{A}_X} \frac{P(X,Y)}{P(Y)} = \frac{P(Y)}{P(Y)} = 1$

[8] We provide here a simplified example of Bayes' rule by analysing the reliability of a COVID-19 test. Although the numbers used here are not far from reality, please refer to official test specifications and legal requirements before applying this example to real life. In general, it is differed then between the test sensitivity and specificity of a test instead of just the reliability.

The probability of Mike being infected if the test is positive $P(a = 1|b = 1)$ can be calculated by first using the definition of conditional probability (2.8):

$$P(a = 1|b = 1) = \frac{P(a = 1, b = 1)}{P(b = 1)}. \tag{2.15}$$

Furthermore, by using the properties of conditional probabilities (2.9), (2.10) and (2.11) we get:

$$P(a = 1|b = 1) = \frac{P(b = 1|a = 1)P(a = 1)}{P(b = 1, a = 1) + P(b = 1, a = 0)} = \frac{P(b = 1|a = 1)P(a = 1)}{P(b = 1|a = 1)P(a = 1) + P(b = 1|a = 0)P(a = 0)}. \tag{2.16}$$

When computing with the given values, one gets:

$$P(a = 1|b = 1) = \frac{0.95 * 0.01}{0.95 * 0.01 + 0.05 * 0.9} = 0.16. \tag{2.17}$$

In conclusion, the probability that Mike is infected is 16%.

Inference Problems

Inverse problems (statistical inference/estimation) are problems in which we want to infer the parameters θ that define a system from some data D. To do so, we use the Bayes formula (2.11):

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}. \tag{2.18}$$

$P(\theta|D)$ is called the *posterior* distribution and is a function of the parameters θ. $P(D|\theta)$ is the *likelihood*, $P(\theta)$ the *prior* and $P(D)$ the *evidence* or *normalization*. It is important to note that the evidence/normalization $P(D)$ does not depend on θ and can be seen as a constant. Its job is to normalize the posterior distribution ($\sum_\theta P(\theta|D) = 1$).

2.3.1 An Example of an Inverse Problem – Decay Constant

Consider a source of unstable particles, which are emitted through a collimator and decay at a distance $x \in \mathbb{R}$ from the source. The random variable x follows the probability distribution (exponential distribution):

$$P(x) = \frac{1}{\lambda}\exp\Big(-\frac{x}{\lambda}\Big). \tag{2.19}$$

Suppose we wish to infer λ^{true}, the "ground truth" value of λ. We set up a detector in front of the source which can measure the value x for a given decay. Unfortunately, the detector only covers a finite range of values, let us say, it can only measure decays between $1 \leq x \leq 20$:

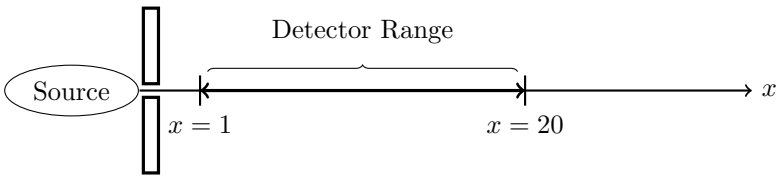


Figure 4: A figure depicting the source of unstable particles, which travel in a straight line and decay at distance x from the source. Notice how we have depicted the range of the detector, from $x = 1$ to $x = 20$.

If we detect n measurements of x, call this $X = \{x_1, x_2, ..., x_n\}$, what is the best estimate of λ? Assume that all these measurements are independent. This is an *inverse* problem.

- Naive Approach (incorrect): One might suggest that the best prediction of λ is simply the mean of all x_i. That is, $\lambda^{\text{naive}} = 1/n \sum_{i=1}^{n} x_i = \bar{X}$. The argument for this approach is that:

  $$\int_0^\infty xP(x)dx = \lambda = \bar{x}.$$

  However, this method is actually *incorrect* as a result of the finite range of the detector. For example, if $\lambda^{\text{true}} = 30$ it would be impossible to obtain $\bar{X} = 30$ since the detector does not include $x = 30$ in its range. There is a better way.

- Correct Approach: A better method is to use Bayesian Inference. Given data $X \in \mathbb{R}^n$, we would like to calculate $P(\lambda|X)$, which we recognize as an inverse problem. We start out by writing what we know. First, we know the likelihood, $P(x_i|\lambda)$, or the probability of getting a measurement $x_i \in X$, assuming we have λ given:

$$P(x_i|\lambda) = \frac{1}{\lambda Z(\lambda)} e^{-x_i/\lambda} \quad \text{for } i = 1, 2, ..., n. \tag{2.20}$$

Here, we have introduced $Z(\lambda)$ as an normalization factor, which we use to normalize the probability. Let us normalize the likelihood now. To do this, we integrate over the detector range, $]1, 20[$:

$$\begin{aligned} Z(\lambda) &= \int_1^{20} \frac{1}{\lambda} e^{-x_i/\lambda} dx_i \\ &= \Big(e^{-1/\lambda} - e^{-20/\lambda} \Big). \end{aligned}$$

Now, the joint probability to obtain all n measurements in X is the product of all the individual likelihoods, because all measurements are independent:

$$P(X|\lambda) = \prod_{i=1}^{n} P(x_i|\lambda) = \Big(\frac{1}{\lambda Z(\lambda)} \Big)^n \exp\Big(-\frac{1}{\lambda} \sum_{i=1}^{n} x_i \Big). \tag{2.21}$$

Finally, we write the Bayes Formula, using the likelihood above:

$$P(\lambda|X) = \frac{P(\lambda)P(X|\lambda)}{P(X)} = \frac{P(\lambda)}{P(X)} \Big(\frac{1}{\lambda Z(\lambda)} e^{-\bar{X}/\lambda} \Big)^n, \tag{2.22}$$

where we have denoted $\bar{X} := 1/n \sum_{i=1}^{n} x_i$ the mean of the data. A good estimate of λ^{true} is the value which maximizes the posterior $P(\lambda|X)$. Since we wish to maximize $P(\lambda|X)$ with respect to λ, constants no longer matter. We know that $P(X)$ is constant in λ, so we choose to ignore it. The problem seems to be $P(\lambda)$, the prior, because it is a function of λ. We do not have any extra information to predict its form, so we make an assumption. First, we assume that the prior is *not a function of n*. Then, we argue that the posterior is dominated by the likelihood in the regime $n \gg 1$, and that the prior will have little or no impact on the maximization of $P(\lambda|X)$. This is due to the exponent of n in the expression for likelihood, which dominates for $n \gg 1$. This estimator is called the "**Maximum Likelihood Estimator**":

$$\hat{\lambda} = \text{argmax}_\lambda \Big(P(X|\lambda) \Big) = \text{argmax}_\lambda \Big(\frac{1}{\lambda Z(\lambda)} e^{-\bar{X}/\lambda} \Big), \tag{2.23}$$

where $\hat{\lambda}$ is the estimate of the ground truth decay constant, λ^{true}. One may find the value of $\hat{\lambda}$ numerically, by maximizing the likelihood with respect to λ. Another method to obtain $\hat{\lambda}$ is to find it graphically, by plotting the likelihood as shown in Figure 5. The result for $\hat{\lambda}$ does not have to lie within the detector range, and is a better estimate of λ^{true} than simply the mean, $\bar{X}$.

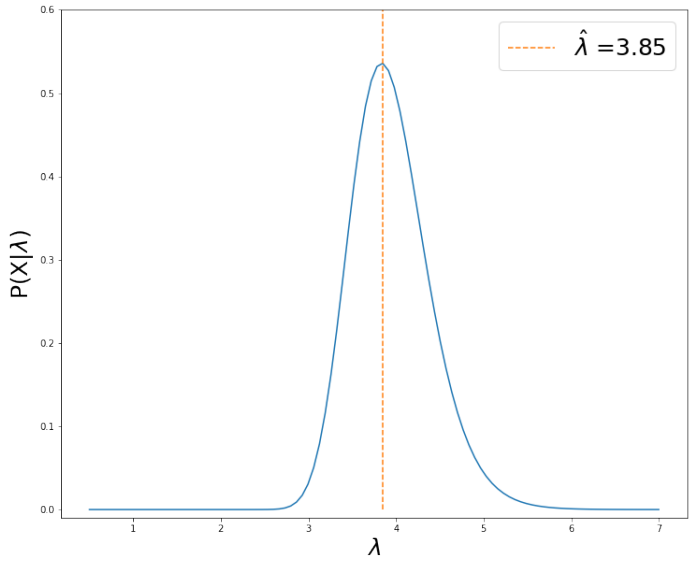


Figure 5: Plot of numerical simulation of the problem generated with ground truth $\lambda^* = 4.0$ and 100 samples.

3 Lecture 3: Linear regression through the lenses of statistical inference 8.10.

Students designated to write this section: Peng Kanlai, Pérez Ortiz Rodrigo, Pohle Paul Timo, Rey Anthony Bernard, Rodriguez Mateos Borja, Roy Camille René Charles
Coordinator:
TA: Federica (TA approved)
Students who actually contributed: Peng Kanlai, Pérez Ortiz Rodrigo, Pohle Paul Timo, Rey Anthony Bernard, Roy Camille René Charles

The least square method is widely used in data processing and error estimation. So far, it is natural to ponder the questions: *Why least squares?* or *Why the regularization?*

3.1 Probabilistic view of Least Squares

Given the data $X = (\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)^T \in \mathbb{R}^{n\times d}$ and labels $y = (y_1, y_2, \ldots, y_n)^T \in \mathbb{R}^n$ all independent and identically distributed (i.i.d.), the Probabilistic Least Squares model considers given data X while the labels y are synthetically generated according to ground truth weight $w^* \in \mathbb{R}^d$ and some noise ξ, giving:

$$y_\mu = \sum_{i=1}^{d} X_{\mu i}\, w_i^* + \xi_\mu, \quad \forall \mu \in \mathbb{N}_1^n. \tag{3.1}$$

The ground truth w^* is sampled from a Gaussian distribution $\mathcal{N}(0, \sigma)$:

$$P(w^*) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma}(w_i^*)^2}. \tag{3.2}$$

The same for the noise ξ_μ: $\forall \mu = 1, 2, \ldots, n$ distributed according to the Gaussian measure:

$$P(\xi_\mu) = \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}(\xi_\mu)^2}. \tag{3.3}$$

It can be noted that noise and weights can be sampled from another distribution, however Gaussian distribution provides a direct mapping to least squares and ridge regularization and therefore gives a better intuition of the choice of the regularization and the model.

There are two principal goals: Given the synthetic dataset $\mathscr{D} = \{X, y\}$ and without knowing the ground truth w^* or the noise ξ:

(a) Prediction: Based on the observed data, predict the label y_{new} associated to a previously unseen example $\boldsymbol{X}_{\text{new}}$.

(b) Estimation: Infer or estimate the ground truth parameters w^*.

Note that if we solved the estimation problem we can solve also the prediction by using the estimator $\hat{w}$ to make predictions on new and previously unseen samples as $y_{\text{new}} = \sum_{i=1}^{d} X_{\text{new,i}}\, \hat{w}_i$. We will thus in the following focus on the estimation problem.

The model being probabilistic, the *Bayes' rule* allows to estimate w^*, i.e. the probability of a certain realization of the coefficients w can be written using the Bayes' rule:

$$\underbrace{P(\overbrace{w}^{\text{Unknown}} \mid \overbrace{X, y}^{\text{Observed}})}_{\text{Posterior}} = \underbrace{\frac{1}{P(y|X)}}_{\text{Evidence}} \underbrace{P(y|w, X)}_{\text{Likelihood}} \underbrace{P(w|X)}_{\text{Prior}}. \tag{3.4}$$

Following from the previous assumptions of the model, the likelihood describes the probability of having observed a certain label given the input data X and the coefficients w. The likelihood is assumed to be Gaussian since the labels are Gaussian random variables. Their mean is given by the scalar product between the input and the weights and the standard deviation by the standard deviation of the noise:

$$P(y|w, X) = \prod_{\mu=1}^{n} \frac{1}{\sqrt{2\pi\Delta}} \exp\left\{ \left[-\frac{1}{2\Delta} \underbrace{\left(y_\mu - \sum_{i=1}^{d} X_{\mu,i} w_i \right)^2}_{\xi_\mu^2} \right] \right\}. \tag{3.5}$$

The ground truth is generated by a Gaussian distribution and the prior represents the initial guess on the distribution from which the estimator $\hat{w}$ should be sampled. From the definition of the models the prior is assumed Gaussian:

$$P(w|X) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma}(w_i)^2}, \tag{3.6}$$

The posterior represents the probability of having coefficients w given the input data, so it updates the original guess given by the prior about the model in the light of the input data. Finally, the evidence can be thought at this stage just as a normalization constant, which ensures that the posterior is normalized to one. In lecture 8, however, it will become important for hyper-parameter learning in cases where the model of the ground truth and of the noise is not given, but have to be found.

We are now ready to discuss the **maximum likelihood estimator** $\hat{w} \equiv \hat{w}_{\text{ML}}$ corresponding to the above probabilistic model. The maximum likelihood estimator is chosen such that the likelihood $P(y|w, X)$ is maximized. This means that one selects the parameters w such that the labels y are most likely to be observed given X. For practical reasons it is common to maximize the logarithm of the initial function, which does not change anything due to the logarithm being a strictly monotonous function. By looking at Eq.(3.5) and ignoring all constant, we can see that the *log-likelihood* is simply the sum of all exponents. We get therefore the following:

$$\hat{w}_{\text{ML}} = \underset{w}{\text{argmax}} \left[P(y|w, X) \right] = \underset{w}{\text{argmax}} \left[\log \left(P(y|w, X) \right) \right] = \tag{3.7}$$

$$= \underset{w}{\text{argmax}} \left[-\frac{1}{2\Delta} \sum_{\mu=1}^{n} \left(y_\mu - \sum_{i=1}^{d} X_{\mu i} w_i \right)^2 \right] = \underset{w}{\text{argmin}} \left[\sum_{\mu=1}^{n} \left(y_\mu - \sum_{i=1}^{d} X_{\mu i} w_i \right)^2 \right]. \tag{3.8}$$

This expression is the ordinary least squares method that was used earlier. From this, one can see that the least squares method comes out naturally if the data contains Gaussian additive noise and if one wants to maximize the likelihood.

We can find the expression for the **maximum a posteriori estimator** $\hat{w} \equiv \hat{w}_{\text{MAP}}$. For this estimator we want to choose the $\hat{w}$, that is most likely, given the available data X and y. In other words, we want to find:

$$\hat{w}_{\text{MAP}} = \underset{w}{\text{argmax}}\, P(w|X, y). \tag{3.9}$$

Because we do not know $P(w|X,y)$ explicitly, Bayes' rule is used to express the posterior in terms of the likelihood and the prior, which we both know. The usage of their expressions from Eq.(3.5) and (3.6), and carrying out the products lead to:

$$\begin{aligned} P(w|X,y) &= \frac{P(y|w,X)P(w|X)}{P(y|X)} \\ &= \frac{1}{P(y|X)} \cdot \frac{1}{(2\pi\Delta)^{\frac{n}{2}}} \exp\left(-\frac{1}{2\Delta}\sum_{\mu=1}^{n}\left(y_\mu - \sum_{i=1}^{d} X_{\mu i} w_i\right)^2\right) \cdot \frac{1}{(2\pi\sigma)^{\frac{d}{2}}} \exp\left(-\frac{1}{2\sigma}\sum_{i=1}^{d} w_i^2\right). \end{aligned} \tag{3.10}$$

Thinking about the maximization of the logarithm of the function instead of the function itself gives us a much clearer picture. By using (3.10), taking the natural logarithm, ignoring all constants outside the exponential, and pushing the constant inside the exponential on one term the result can be easily found as being the following:

$$\hat{w}_{\text{MAP}} = \underset{w}{\text{argmax}}\left[P(w|X,y)\right] = \underset{w}{\text{argmax}}\left[\log\left(P(w|X,y)\right)\right] = \underset{w}{\text{argmin}}\left[\sum_{\mu=1}^{n}\left(y_\mu - \sum_{i=1}^{d} X_{\mu i} w_i\right)^2 + \frac{\Delta}{\sigma}\sum_{i=1}^{d} w_i^2\right]. \tag{3.11}$$

Here we recognize that with $\lambda = \frac{\Delta}{\sigma}$ the least squared method with ℓ_2 regularization, which was already used in previous sections, is derived. Therefore we can conclude, that the introduction of a Gaussian prior that is independent and identically distributed for all w_i yields the ℓ_2 regularization[9] if the posterior probability is maximized.

3.2 Linear Regression and Inverse Problems in Signal Processing

Now, we want to talk about the relation between linear regression and inverse problems in signal processing and imaging processing. To illustrate that, two concrete examples will be presented.

X-Ray Tomography

The first example is the X-ray Tomography. In X-ray computed tomography, we measure with a detector the intensity of X-rays after crossing a sample. Like this the absorption of X-ray in the sample can be calculated. In Figure 6, one

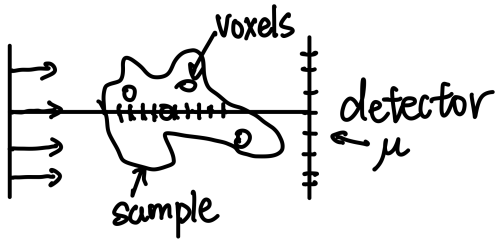


Figure 6: X-ray

can see that the X-rays penetrate the sample from left to right, and finally reach the detector. Here $\mu \in \{1,2,\ldots,n\}$ refers to the index related to the detector cells, n denotes the total number of pixels of all detectors. Normally when computed tomography is done, the source is turned around the sample so there would be measurements from different angles, which would thus lead to different indices μ_k. This will be ignored here. The different indices $i = \{1,2,\ldots,d\}$ refer to the voxels of the sample, d is the total number of pixels in the map of the sample. In this case w^* is considered as the wanted signal (absorption map). Therefore, a measurement matrix $X_{\mu i}$ can be defined. This matrix represents the voxels i which influence the X-rays being later on detected by the μ-th cell of the detector. What the detector measures is the beam intensity (and not the absorption) with noise at position μ. The absorption will here be assumed to be linear with the penetration depth. The noise is assumed to be Gaussian distributed. Therefore we can write a model with y_μ the values the n detectors actually measure, ξ_μ the noise, $X_{\mu i}$ the measurement matrix and w_i^* the absorption map.

$$y_\mu = \sum_{i=1}^{d} X_{\mu i} w_i^* + \xi_\mu \,. \tag{3.12}$$

Reconstructing the image thus equals to inferring w^* from $\{y, X\}$. The model let us find from the intensity measurement the absorption map of the sample.

[9] Models with this type of regularisation are also called *ridge regression* or *Tikhonov regularization*.

Nuclear Magnetic Resonance Spectroscopy

Another example is the nuclear magnetic resonance spectroscopy (NMR), also called magnetic resonance imaging (MRI) when images are produced. It is similar to the first example. Variable $y \in \mathbb{R}^n$ is what is measured by the detectors, $X_{\mu i} \in \mathbb{R}^{n \times d}$ the measurement matrix, $\xi \in \mathbb{R}^n$ the noise and $w^* \in \mathbb{R}^d$ the signal (map of the tissue sample). In a nutshell: Again, not the signal of interest w^* is measured, but a linear transformation of it.

$$y_\mu = \sum_{i=1}^{d} X_{\mu i} w_i^* + \xi_\mu \,. \tag{3.13}$$

In the case of NMR the matrix X corresponds to a Fourier transform in order to go from time to frequency domain. In simple words: Reconstructing images in MRI is again an inverse problem which is mathematically and algorithmically equivalent to linear regression – inferring w^* from $\{y, X\}$. However, actual MRI with all nowadays features and the baseline theory is a bit more complicated than that.

One important remark is that in both applications, the goal is not making any predictions but lays in inferring w^*. We want to reconstruct the image or a signal of interest instead of predicting some sort of new "datapoints".

3.3 Generalized Linear Model

Previously, we considered a probabilistic model with Gaussian prior and noise. This leads to a direct mapping with least squares and ridge regression. One defines a generalised probabilistic model that takes input $X \in \mathbb{R}^{n \times d}$ and generates the labels by sampling them from the generic conditional probability distribution of $y \in \mathbb{R}^n$ given X and $w^* \in \mathbb{R}^d$, which will tell us something about the noise:

$$P_{\text{out}}(y_\mu \mid \sum_{i=1}^{d} X_{\mu i} w_i^*) \,. \tag{3.14}$$

Like before, assume that w_i^* are the ground truth values of parameters. This time, however, the prior $P_{\vec{w}}(w)$ is completely generic (cf. with Eq.(3.6) where the prior is not generic):

$$P_{\vec{w}}(w) = \prod_{i=1}^{d} P_w(w_i) \,. \tag{3.15}$$

Note that this is a special case where the prior is independent of X and it is assumed to be factorized over the different components (i index). From this it is clear to see that this formulation allows to describe other tasks than regression, such as classification (see Lecture 5).

The goal is the same: With given data X, y and a given generalised linear model (GLM) P_{out} and $P_{\vec{w}}$, one can estimate $\hat{w}$ in order to infer the ground truth and then predict on unseen samples. As before, we can thus write down the posterior distribution as product over all samples of the likelihood times the prior:

$$\begin{aligned} P(w \mid X, y) &= \frac{1}{P(y \mid X)} P(y \mid w, X) P(w, X) \\ &= \frac{1}{P(y \mid X)} \prod_{\mu=1}^{n} P_{\text{out}}(y_\mu \mid X_\mu \cdot w) \prod_{i=1}^{d} P_w(w_i) \,. \end{aligned} \tag{3.16}$$

where in the last row it is assumed that the samples can be treated independently. From the posterior the **maximum a posteriori estimator** (MAP) is computed, in this case also called the **empirical risk minimisation** (ERM) and written as:

$$\hat{w}_{\text{ERM}} \equiv \hat{w}_{\text{MAP}} = \underset{w}{\operatorname{argmin}} \, \mathscr{L}(w) \tag{3.17}$$

with the full loss $\mathscr{L}(w)$:

$$\begin{aligned} \mathscr{L}(w) &= -\frac{1}{n} \sum_{\mu=1}^{n} \ln P_{\text{out}}(y_\mu | X_\mu \cdot w) - \frac{1}{n} \sum_{i=1}^{d} \ln P_w(w_i) \\ &\equiv \frac{1}{n} \sum_{\mu=1}^{n} l(y_\mu, X_\mu \cdot w) + \frac{1}{n} \lambda \sum_{i=1}^{d} r(w_i) \end{aligned} \tag{3.18}$$

having introduced the *one-sample loss* l and the regularization r. The same happens for the **maximum likelihood estimator** that becomes:

$$\hat{w}_{\text{ML}} = \underset{w}{\text{argmin}} \sum_{\mu=1}^{n} l(y_\mu, X_\mu \cdot w)\,. \tag{3.19}$$

4 Lecture 4: Other losses and regularizations, sparsity 15.10.

Students designated to write this section: Mitais Charles Edmond Alfred, Moawad Alix Marie Gabrielle, Monnet Nathan Jean, Niggli Loïs, Norberg Astrid Mona Joséphine, Parusa Gian
Coordinator: Parusa Gian
TA: Giovanni (TA approved)
Students who actually contributed: Mitais Charles Edmond Alfred, Moawad Alix Marie Gabrielle, Monnet Nathan Jean, Niggli Loïs, Norberg Astrid Mona Joséphine, Parusa Gian

4.1 Main linear regression methods

In Eq.(3.18) we have defined the loss function we must minimize in order to get the maximal a posteriori (MAP) estimator $\hat{w}$. If we look at the second equality of this equation, we introduced the one-sample loss l (some distance between the labels y and the scalar product between the data points X and some weights w) and r the regularization. We already saw an example of losses like the least squares, which corresponds to: $l(y_\mu, X_\mu \cdot w) = (y_\mu - X_\mu \cdot w)^2$ and an example of regularization like the square regularization where $r(w_i) = w_i^2$. With this loss and the latter regularization, we were able to get an explicit expression for the estimator. However, numerous other losses and regularizations exist and give rise to several other linear regression methods. The main methods we will discuss here are: the ridge regression, the robust regression, and the sparse regression. All these can be derived as maximum a posteriori estimation of special cases of the Generalized Linear model (GLM) introduced previously. Using again the machine learning notation $X = (\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n)^T \in \mathbb{R}^{n\times d}$ with labels $y = (y_1, y_2, \dots, y_n)^T \in \mathbb{R}^n$, this is leading to a linear regression that minimizes the following loss:

$$\mathscr{L}(w) = \frac{1}{n}\sum_{\mu=1}^{n} l(y_\mu, X_\mu \cdot w) + \lambda \frac{1}{n}\sum_{i=1}^{d} r(w_i)\,. \tag{4.1}$$

for various choices of the loss l and regularization r. We will also look at an universal iterative algorithm that can be used to minimize the loss function which is called *Gradient Descent* and some variants of it.

4.1.1 Ridge regression

In the most basic version the GLM model assumes that the conditional distribution P_{out} of the labels y given the inputs X and the parameters w are both Gaussian:

$$P_{\text{out}}(y_\mu | X_\mu \cdot w) = \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}(y_\mu - X_\mu \cdot w)^2}\,. \tag{4.2}$$

By computing the one-sample loss function l which is minus the logarithm of the likelihood, we obtain:

$$\begin{aligned} l(y_\mu, X_\mu \cdot w) &= -\log P_{\text{out}}(y_\mu | X_\mu \cdot w) \\ &= (y_\mu - X_\mu \cdot w)^2, \end{aligned} \tag{4.3}$$

which corresponds to the least squares. A regression using such a loss is called *ridge regression*. We also assume that the prior on w^* behaves as a Gaussian:

$$P(w|X) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma} w_i^2}\,. \tag{4.4}$$

This gives us the following expression for the regularization: $r(w_i) = -\log P_w(w_i) = w_i^2$ which corresponds to L_2 regularization.

By computing the posterior $P(w|X,y)$ for specific value of $P_{\text{out}}(y_\mu|X_\mu \cdot w)$ and $P(w|X)$ given by Eq.(4.2) and (4.4) we get the maximum a posteriori estimator (MAP) $\hat{w}$ by using Eq.(3.17) and (3.18). It has the form:

$$\hat{w} = \left(X^T X + \lambda \mathbb{I}\right)^{-1} X^T y\,, \quad \lambda = \frac{\Delta}{\sigma}\,. \tag{4.5}$$

4.1.2 Robust Regression

Ridge regression uses a loss function (the negative log-likelihood) which represents a second-degree polynomial. There is actually another type of loss function that uses *Laplace* distribution that has heavier tails so that the outliers are less penalized. The idea is to use a probability density that weights the outliers only linearly in the exponential. We thus assume that the noise follows a Laplace distribution from which we can derive the likelihood given by:

$$P_{\text{out}}(y_\mu | X_\mu \cdot w) = \frac{\gamma}{2} e^{-\gamma |y_\mu - X_\mu \cdot w|} \, . \tag{4.6}$$

The maximum likelihood estimation for this model thus leads to a loss function given by:

$$l(y_\mu, X_\mu \cdot w) = \gamma |y_\mu - X_\mu \cdot w| \tag{4.7}$$

which is called **mean absolute error loss** (MAE). A Regression that uses such a one-sample loss is called a *robust regression*.

The interpretation for this robustness is as follows. Consider what would be the effect of an outlier on ridge regression compared to robust regression? Suppose we know the correct value of w (the ground truth w^*), but the data contains an outlier x_o, y_o (i.e. a data point which is very far from $y = wx$). Then, if we compute as in ridge regression the square loss of w^* on the training set there will be a huge term $(y_o - w^* x_o)^2$ in the training loss. So, if we minimize the training loss we will get a predictor $\hat{w}$ which is far from w^*. If instead the loss is $|y_o - w^* x_o|$, as in the robust regression, the contribution of the outlier to the training loss will be much smaller and the estimator $\hat{w}$ will be less influenced by the outlier. Therefore, the robust regression puts less weight on the outliers than the ridge regression. This is illustrated on Figure 7.

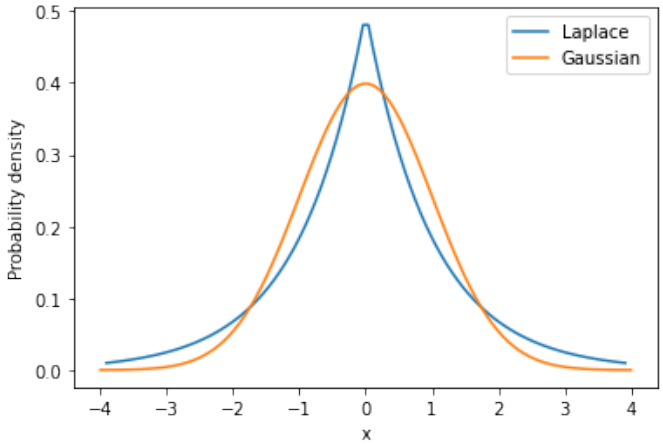


Figure 7: Gaussian versus Laplace distribution

4.1.3 Sparse Regression

Here the noise is the same as in the ridge regression, cf. Eq.(4.2), but this time the prior follows a Laplace distribution $P(w_i) = \frac{\lambda}{2\Delta} e^{-\lambda |w_i| / \Delta}$. Therefore, the total loss function is given by:

$$\mathscr{L}(w) = \frac{1}{n} \sum_{\mu=1}^{n} (y_\mu - X_\mu \cdot w)^2 + \lambda \frac{1}{n} \sum_{i=1}^{d} |w_i| \, . \tag{4.8}$$

This loss is called the **least absolute shrinkage and selection operator** (LASSO). Due to the sum of absolute values of w_i the regularization terms is called a L_1 regularization and corresponds to the *sparse linear regression*.

To better understand what changes by using a Laplace distribution instead of Gaussian distribution in the regularization term, let us look at the following contour plots. For the sake of plotting, here we assume the data is two-dimensional, such that $\vec{w} = (w_1, w_2)$.

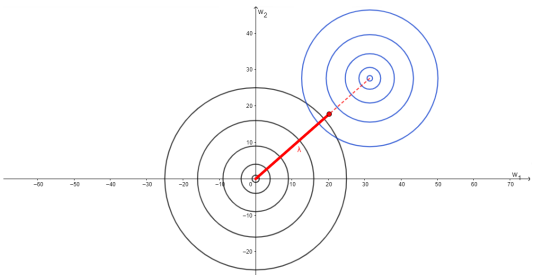


Figure 8: L_2 regularization in 2D.

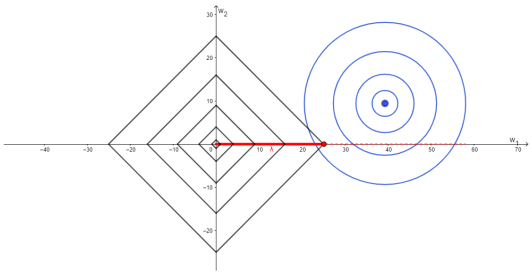


Figure 9: L_1 regularization in 2D.

The LASSO loss can be seen as constrained Lagrange minimization, where the loss function is minimized under a constraint on the norm of w. In L_2 regularization, the constraint on the norm gives a quadratic function appearing as black circles in the contour plot of the space parameters (w_1,w_2). The loss is a quadratic function and is represented by its level curves drawn as blue circles in Figure 8. The role of the regularization strength λ is then to shift the point of minimization (along the red segment) by modifying the importance of the constraint on the norm of w in the loss function.

What changes in the L_1 regularization is the equation of the constraint on the norm? Instead of having a circle contour for the constraint as for the L_2 regularization, one has a diamond-like shape in the contour plot due to the absolute value function in the regularization term, as shown in Figure 9. In most directions, due to the geometry, the point of minimization will tend to hit the space of w in a corner. In this specific example, it means that $\hat{w}_2 = 0$. In a more general case, the L_1 regularization induces sparsity of $\hat{w}$, meaning that the estimated $\hat{w}$ will have many components equal to 0.

As a machine learning motivation, this may have an importance in the variable selection and in the interpretation of $\hat{w}$. Indeed, if $\hat{w}_i$=0, then $X_{\text{new},i}$ does not matter for predicting y_{new}. So, it enables the ability to select among the different variables of the problem the ones that are more relevant for an accurate prediction.

In terms of signal processing, the importance of sparse regression is compressed sensing as it can reduce the time and memory needed to reconstruct the signal. Being compressible means, that there exists a basis matrix $G \in \mathbb{R}^{d\times d}$, where d is the dimension of an input, such that,

$$s = Gw \ , \tag{4.9}$$

where s is the input signal vector and w is a sparse vector. In a measurement, the measured signal y can be expressed in terms of the input signal s and the noise ξ accompanying it as,

$$y_\mu = F_\mu \cdot s + \xi_\mu \ , \tag{4.10}$$

where the index μ stands for the index of the sample and F is a matrix representing the internal mechanism of the measurement device. Using Eq.(4.9) and (4.10), one can write:

$$y_\mu = X_\mu \cdot w + \xi_\mu \ . \tag{4.11}$$

Here, $X_\mu = F_\mu G$. This equation simply tells us that there is indeed a way of compressing the measured signal to a sparse vector. However, if one chooses ridge regression, the estimator for w will not be sparse, as already explained in Figure 8. Therefore, one should choose sparse linear regression (LASSO) instead.

4.2 Gradient Descent

Gradient descent is a universal, iterative algorithm that attempts to minimise a function. Thus, it can be used to derive an estimator $\hat{w}$ from the minimization of a loss function. In the following we will first give the analytical expression of the gradient descent and see how we can retrieve it. Then, we will illustrate the gradient descent algorithm with a graphical explanation allowing us a better understanding of the process.

We first initialize the weights[10] $\vec{w} \equiv w$ (sampling them from a random distribution is quite common for instance). We also choose a parameter called learning rate denoted by η. This parameter describes the step size we will take at each iteration of the gradient descent. Then at each step, $\vec{w}$ is updated by subtracting to its previous value the gradient of the loss function at the current point multiplied by the learning rate. The aim is therefore to find the value $\hat{w}$ for which we reach the minimum of the loss function. The iterative process is given $\forall i \in \{1, \dots, d\}$ by:

[10] As the reader may already have understood, in the context of machine learning the vector sign is often omitted to not complicate the notation.

$$\boxed{w_i^{t+1} = w_i^t - \eta \frac{\partial \mathscr{L}(\vec{w})}{\partial w_i}\bigg|_{\vec{w}^t}}\,, \tag{4.12}$$

where the indices t and $t+1$ represent two consecutive iterations while η is the learning rate (also called step size). The question that can be raised now is how to choose the parameter η. If η is too small, we will converge really slowly towards a minimum and the algorithm will be computationally expensive. On the other hand, if it is too large, we might risk missing a minimum of the loss function and then diverge. Another risk is that the calculated loss jumps around the minimum without reaching it, since the step size is too large. Therefore, a trade-off is needed to find an optimum between these two behaviours.

In the following paragraph we will retrieve the expression of the gradient descent algorithm given by Eq.(4.12). To ease the calculations, we will take $d = 1$ so that mathematically correctly $\vec{w} = w$ can be seen as a scalar. A Taylor expansion of the loss function at a given w^t can be computed:

$$\mathscr{L}(w) \approx \mathscr{L}(w^t) + \frac{\partial \mathscr{L}(w)}{\partial w}\bigg|_{w^t}(w - w^t) + \frac{1}{2}\overbrace{\frac{\partial^2 \mathscr{L}(w)}{\partial w^2}\bigg|_{w^t}}^{1/\eta}(w - w^t)^2\,. \tag{4.13}$$

The gradient descent replaces the second derivative by $1/\eta$. The last term of Eq.(4.13) can then be seen as a parabola of width proportional to η and the gradient descent looks for the minimum of this parabola to find the next $w = w^{t+1}$ in the iterative process. The minimum of the parabola is found by taking the derivative of the previous equation (4.13) and set it to 0 :

$$\frac{\partial \mathscr{L}(w)}{\partial w} = 0 = \frac{\partial \mathscr{L}(w)}{\partial w}\bigg|_{w^t} + \frac{1}{\eta}(w - w^t) \quad \Rightarrow \quad w = w^t - \eta \frac{\partial \mathscr{L}(w)}{\partial w}\bigg|_{w^t}\,. \tag{4.14}$$

By understanding that $w \equiv w^{t+1}$ in the conclusion of Eq.(4.14) we finally find the same expression as the one in Eq.(4.12), giving us an iterative algorithm to find the estimator $\hat{w}$. Let us now give a more intuitive explanation of the algorithm.

To understand what is going on in the algorithm, we plot the loss function $\mathscr{L}(w)$ as a function of the weight w in Figure 10. When going from w^t to w^{t+1}, the algorithm approximates the loss function at w^t by a parabola with width proportional to η. The w^{t+1} is chosen to be the minimum value of this parabola, as implied by Eq.(4.14). Here one can clearly see what happens when η is too large. The minimum of the parabola may overpass the global minimum of the loss function $\mathscr{L}(w)$ which then may makes the algorithm diverges or at least not converging. However, choosing η to be too small makes the parabola becomes narrower, implying its minimum to be very close to w^t. This in return makes the progression of the algorithm slow. Therefore, as stated earlier, one needs to take the η value somewhere between these two extremes.

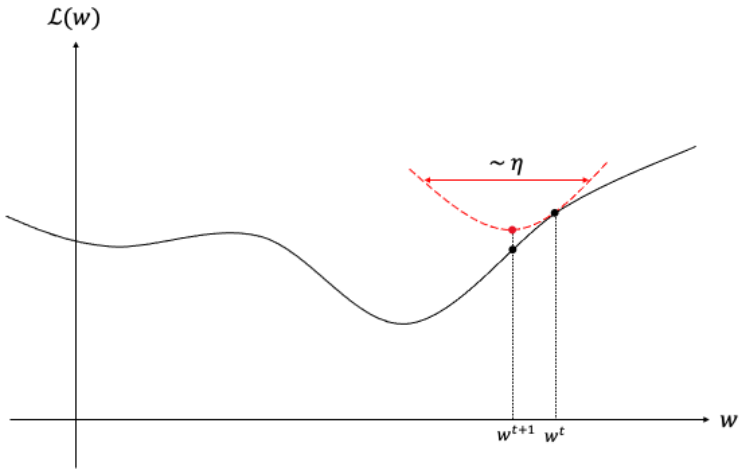


Figure 10: Visualization of the gradient descent algorithm

As with all iterative algorithms, the question is when to stop and how many iterations should be done. We need to wait until the algorithm converges towards a specific value of w which will therefore be the estimator. We often use in Machine Learning an early stopping based on the validation set: If the performance of the model on the validation set starts to degrade then we stop the training, i.e., at the point of smallest error (or at least at the point we assume to be the smallest possible error).

Even if the gradient descent algorithm is running till convergence, it will often reach only a local minimum for non-convex losses. If one wants to reach a global minimum one is required to explore more extensively the exponentially large space of weights w (for example by initializing GD at a different point). Despite this lack of guarantee to reach the global minimum variants, of gradient descent algorithms are widely used for non-convex loss functions in modern machine learning.

Variants of Gradient Descent

In practice, Gradient Descent is not used in ML. Instead, the so called Stochastic Gradient Descent (SGD) is widely used on large datasets.

Stochastic Gradient Descent The derivative of the loss,

$$\frac{\partial \mathscr{L}(w)}{\partial w_i} = \frac{1}{n}\sum_{\mu=1}^{n} \frac{\partial l(y_\mu, X_\mu \cdot w)}{\partial w_i} + \lambda \frac{1}{n}\frac{\partial r(w_i)}{\partial w_i} \,, \tag{4.15}$$

involves a sum that can be very large (possibly millions of samples), making it hard to compute. The idea of SGD is to reduce it by summing only over samples μ that lie in a *minibatch* B_t. The derivative of the loss becomes,

$$\frac{\partial \mathscr{L}(w)}{\partial w_i} = \frac{1}{n}\sum_{\mu\in B_t}^{n} \frac{\partial l(y_\mu, X_\mu \cdot w)}{\partial w_i} + \lambda \frac{1}{n}\frac{\partial r(w_i)}{\partial w_i} \,. \tag{4.16}$$

In each step we use a minibatch, and update the weights as follow:

$$w_i^{t+1} = w_i^t - \eta \frac{\partial \mathscr{L}(w)}{\partial w_i} = w_i^t - \eta \left\{ \frac{1}{n}\sum_{\mu\in B_t} \frac{\partial l(y_\mu, X_\mu \cdot w)}{\partial w} + \lambda \frac{1}{n}\frac{\partial r(w_i)}{\partial w_i} \right\} \,. \tag{4.17}$$

Once every B minibatch have been used, a new random partition is drawn, and the process restarts. A full iteration overall n data points (i.e over every $n/|B_t|$ subset) is called an *epoch*. The calculation of one step is thus much faster. The size of minibatches must be chosen (a typical value could be 100), and then at each iterations a new B_t is chosen at random within the samples. Surprisingly, performing the sum on only the minibatches does not deteriorate the performance (in the sense of test error).

Subgradient descent In the case of LASSO regressions, the loss function is given by Eq.(4.8). To solve the optimisation, we need its derivative with respect to w_i – this time explicitly:

$$n\frac{\partial \mathscr{L}}{\partial w_i} = -2\sum_{\mu}^{n}(y_\mu - X_\mu \cdot w)X_\mu + \lambda \ \mathrm{sign}(w_i). \tag{4.18}$$

However, for $w_i = 0$, the second term does not exist. This is a problem with LASSO regression, as it sets many weights to 0. To circumvent this issue, we use the so-called *subgradient descent*. The idea of this method is to replace the derivative $\frac{\partial r(w)}{\partial w_i}$ whenever it does not exist by any value inside $\left[\frac{\partial r(w)}{\partial w_i}\Big|_{w_i^-}, \frac{\partial r(w)}{\partial w_i}\Big|_{w_i^+} \right]$, the set of existing values of the derivative. For LASSO regression, this set reduces to $[-1, +1]$.

The iterative process is the same as Eq.(4.12). The only difference is in the derivative of the loss, which becomes,

$$\begin{aligned} \frac{\partial |w|}{\partial w_i} &= \begin{cases} +1, & \text{if } w_i > 0 \\ -1, & \text{if } w_i < 0 \end{cases} \,, \\ \frac{\partial |w|}{\partial w_i} &\in [-1, +1], \quad \text{if } w_i = 0 \,. \end{aligned} \tag{4.19}$$

Newton and Second Order Methods Instead of replacing the second derivative in Eq. (4.14) by some constant, this method actually tries to estimate it.

Another idea is to add momentum, i.e., memory of the direction we are following in parameter space. This is another way of putting in the second derivative. In practice it means adding a term w^{t-1} to the iterative process. This procedure helps escaping local minima when minimizing the loss function. One example is ADAM: **Adaptive Moment Estimation**, where η can changed from iteration to iteration.

5 Lecture 5: Linear Classification 22.10

Students designated to write this section: Michalski Elisa Magdalena, Mejean Maïlys Joséphine, Méan Baptiste Maurice, Mauron Linda, Massard Fanny, Martres Delphine Renée Evelyne
Coordinator: Martres Delphine
TA: Alessandro (TA approved)
Students who actually contributed: Michalski Elisa Magdalena, Mejean Maïlys Joséphine, Méan Baptiste Maurice, Mauron Linda, Massard Fanny, Martres Delphine Renée Evelyne

Reminder: Let us remind the types of machine learning problems we have encountered so far. Supervised learning can be split in two different ways. On the one hand, depending on what we are trying to do:

- inference/estimation: Find the best values of the weights, i.e., give w.
- prediction: Be able to predict the outcome from a sample, i.e., get y_{new} from X_{new}.

On the other hand, depending on the kind of outcome value:

- regression if $y_\mu \in \mathbb{R}$ (up to now)
- classification if y_μ is categorical, in $\mathbb{Z}$, $\mathbb{N}$, e.g. $y_\mu \in \{\pm 1\}$ or $\{1, 2, \dots, 9\}$ (now).

5.1 Linear Classification with 2 classes

Example: 2-dimensional simple case. Consider the case, as shown on Figure 11, where to each point, a label is associated:

$$\begin{cases} \times : y_\mu & = +1 \\ \circ : y_\mu & = -1 \end{cases} \tag{5.1}$$

Two sets of points are set to be linearly separable if there exists at least one line that separates all the elements from one set to the other. This can be extended to a general dimension p if there exists at least one hyperplane that separates the two sets. One can see that the data in Figure 11 is linearly separable.

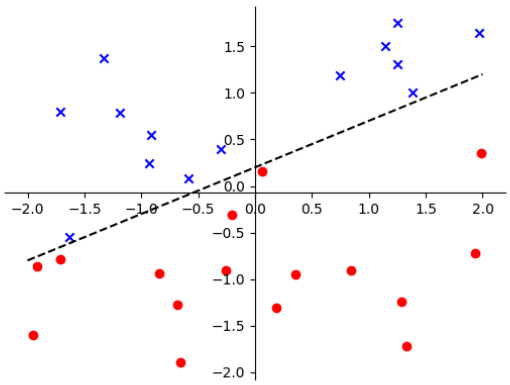


Figure 11: Example of 2-dimensional linear classification with linear separator.

In this example, the dimension is of the entire space is $p = 2$ and therefore, each point of the dataset resides in the p-dimensional space is denoted by $z^{(\mu)} = (z_1^{(\mu)}, z_2^{(\mu)})$ with $\mu \in \{1, 2, \ldots, n\}$. The equation of the separator (hyperplane) is given by $0 = b + \sum_{i=1}^{p} \tilde{w}_i z_i$. When returning to matrix-formulation the hyperplane can be written as:

$$\tilde{X} \cdot w = 0, \tag{5.2}$$

with $\tilde{X} = (z_1, z_2, 1)$, where (z_1, z_2) is a random point of the hyperplane, and $w = (\tilde{w_1}, \tilde{w_2}, b)^T$.

Therefore, the number of coefficients is $d = p + 1$. We denote by $X \in \mathbb{R}^{n \times d}$ the data (the n sample points in the p dimensional space), $w \in \mathbb{R}^d$ the weights and the labels $y \in \mathbb{R}^n$ such that $y_\mu \in \{\pm 1\}$. The considered quantities are:

$$w = \begin{pmatrix} \tilde{w}_1 \\ \tilde{w}_2 \\ b \end{pmatrix}, \quad X = \begin{pmatrix} z_1^{(1)} & z_2^{(1)} & 1 \\ \vdots & \vdots & \vdots \\ z_1^{(n)} & z_2^{(n)} & 1 \end{pmatrix}, \quad y_\mu \in \{\pm 1\}. \tag{5.3}$$

One can solve the problem by finding w such that:

$$\begin{aligned} X_\mu \cdot w > 0 &\iff y_\mu = +1\,, \\ X_\mu \cdot w < 0 &\iff y_\mu = -1\,. \end{aligned} \tag{5.4}$$

Which is the same as aiming $y_\mu = \text{sign}(X_\mu \cdot w)$. But, what would be a good loss function to minimize to find such a w? In fact also linear classification can be seen as a special case of the minimization of the loss Eq.(4.1). Remember that $\mathscr{L}(w) = \sum_\mu^n l(y_\mu, X_\mu \cdot w)$. Common choices of the one-sample loss $l(y_\mu, X_\mu \cdot w)$ include the following:

- Zero-one loss:
$$l_{01}(y_\mu, X_\mu \cdot w) = \frac{1}{2}[1 - y_\mu \text{sign}(X_\mu \cdot w)], \tag{5.5}$$
which corresponds to counting the number of mismatches (i.e., the terms of the sum are +1 for misclassified points and 0 for well-classified points). The estimator is $\tilde{w} = \text{argmin}_w \mathscr{L}(w)$. Remember that in general the minimisation problem is solved using gradient descent. This loss has therefore a clear disadvantage since it is not suitable for gradient descent minimisation as $\mathscr{L}(w)$ has zero derivative almost everywhere.

- Least-square loss :
$$l_{\text{LS}}(y_\mu, X_\mu \cdot w) = (y_\mu - X_\mu \cdot w)^2 \tag{5.6}$$
We have already extensively discussed the square loss and know the explicit solution for the estimator $\hat{w} = (X^T X + \lambda \mathbb{I})^{-1} X^T y$ if the part in brackets is invertible and λ denotes the regularisation strength. This loss function penalizes also correctly classified points. For instance, even if $X_\mu \cdot w$ is correctly positive but far away from 1 (for instance 10), there will be a penalization. Even though this loss function does not seem good, in practice it is not such a bad choice.

- A loss that would only penalize misclassified points could read
$$l_{\text{unnamed}}(y_\mu, X_\mu \cdot w) = \max(0, -y_\mu (X_\mu \cdot w)) \tag{5.7}$$
Indeed, if $y_\mu \cdot (X_\mu \cdot w) \geq 0$, then $\max(0, -y_\mu \cdot (X_\mu \cdot w)) = 0$; if $y_\mu \cdot (X_\mu \cdot w) < 0$, then $\max(0, -y_\mu \cdot (X_\mu \cdot w)) = |y_\mu \cdot (X_\mu \cdot w)|$. Let us now check the gradient. If a point is misclassified, the loss is linear in w and the gradient is non-zero. If a point is, however, correctly classified, the corresponding gradient will be zero.

- Hinge loss :
$$l_{\text{hinge}}(y_\mu, X_\mu \cdot w) = \max(0, 1 - y_\mu (X_\mu \cdot w)), \tag{5.8}$$
which encourages a margin, i.e., not only misclassified points are penalized but also data points that are close to the linear separator (hyperplane). This loss thus often leads to more robust classification.

- Logistic loss
$$l_{\text{logistic}}(y_\mu, X_\mu \cdot w) = \ln\Big(1 + e^{-y_\mu (X_\mu \cdot w)}\Big). \tag{5.9}$$
The logistic loss has the same behaviour as the hinge loss at $\pm\infty$, but it is smooth at $0 = y_\mu (X_\mu w)$.

It should be noted that one has to be careful and add regularisation, otherwise the weights can explode to infinity in logistic regression.

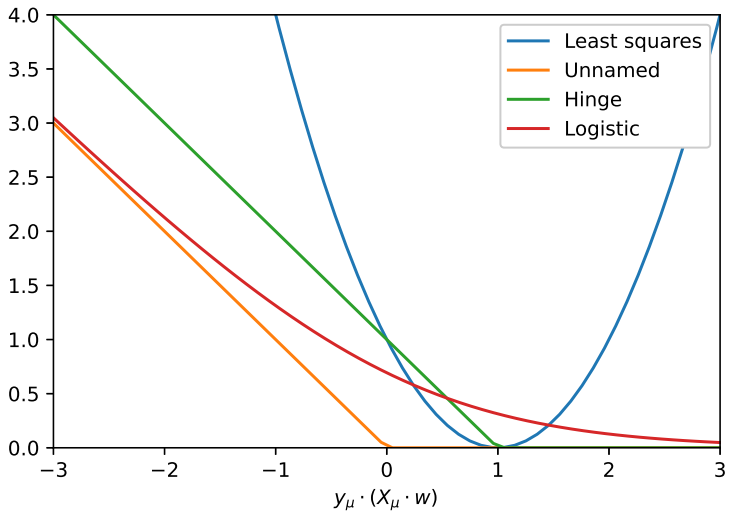


Figure 12: Visualisation of the loss functions

Why Logistic Loss?

Let's consider an example to get an intuition about logistic regression from Bayesian inference. We want to write the a probabilistic model $P_{\text{out}}(y_\mu|w^* \cdot X_\mu)$, and derive the logistic loss from it. The ground truth will be denoted with w^*. We will consider a concrete example where $y_\mu = +1$ means that the exam is passed and $y_\mu = -1$ means that the exam is failed, with $\mu = 1, ..., n$ corresponding to the different students of the class. Let $X_{\mu 1}$ be the number of hours studied, $X_{\mu 2}$ the number of attended lectures, $X_{\mu 3}$ the name of the student, $X_{\mu 4}$ the grade obtained in Statistical Physics 2, and so on. There are d such variables in total. Some of them look relevant, others do not.

Let us assume the following conditional probability distribution:

$$\begin{aligned} P_{\text{out}}(y_\mu|w^* \cdot X_\mu) &= \frac{\exp\left(y_\mu \sum_{i=1}^d X_{\mu i} w_i^*\right)}{\exp\left(\sum_{i=1}^d X_{\mu i} w_i^*\right) + \exp\left(-\sum_{i=1}^d X_{\mu i} w_i^*\right)} \\ &= \frac{1}{1 + \exp\left(-2y_\mu \sum_{i=1}^d X_{\mu i} w_i^*\right)} . \end{aligned} \tag{5.10}$$

Notice that we normalized the probability using the fact that y_μ has only two possible values, respectively $+1$ and -1. The term $\sum_{i=1}^d X_{\mu i} w_i^*$ represents some kind of overall capacity to pass the lecture. For example, we are definitely hoping that one's name does not influence his capacity of passing the lecture, this means that w_3^* should be very close to zero. From this model, we will generate the data, i.e., the y_μ. Using Eq.(5.10) we take the product of the $P_{\text{out}}(y_\mu|w^* \cdot X_\mu)$ over all the students and maximize its logarithm to get the maximum log-likelihood estimator:

$$\begin{aligned} \hat{w}_{\text{ML}} &= \underset{w}{\text{argmax}} \left[\log \prod_{\mu=1}^n P_{\text{out}}(y_\mu|w^* \cdot X_\mu)\right] \\ &= \underset{w}{\text{argmax}} \left[-\sum_{\mu=1}^n \log\left(1 + \exp\left(-2y_\mu \sum_{i=1}^d X_{\mu i} w_i\right)\right)\right] \\ &= \underset{w}{\text{argmin}} \left[\sum_{\mu=1}^n \log\left(1 + \exp\left(-2y_\mu \sum_{i=1}^d X_{\mu i} w_i\right)\right)\right] . \end{aligned} \tag{5.11}$$

We recognize the logistic loss function (Eq.(5.9)) with an additional constant in the exponential. This factor of 2 can, however, be included in the weights and does not matter. The logistic loss is therefore retrieved from the initial assumption on P_{out} (Eq.5.10), which is actually very reasonable. As shown on Figure 13, it implies that the larger $w^* \cdot X_\mu$ (i.e. the overall capacity to pass the exam), the larger $P(y_\mu = 1)$.

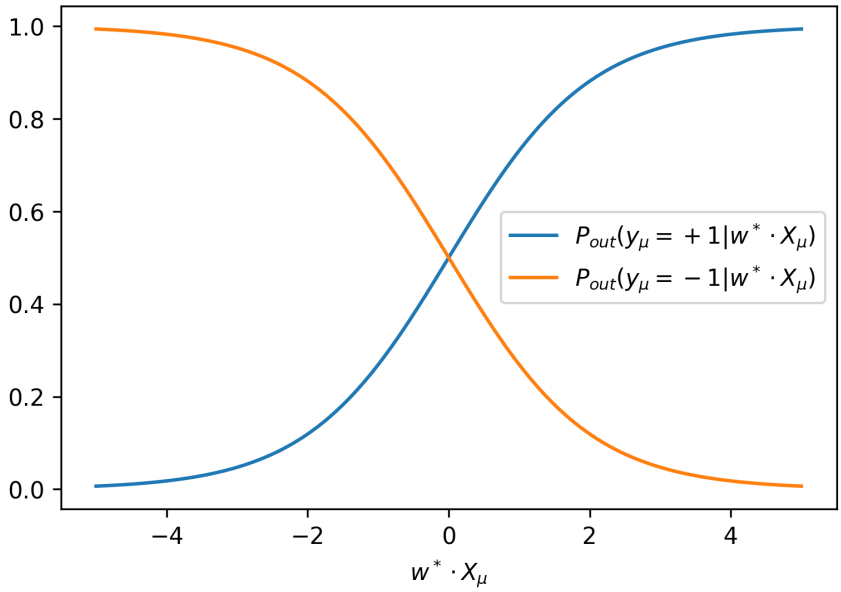


Figure 13: Probabilistic model $P_{\text{out}}(y_\mu|w^* \cdot X_\mu)$ leading to the logistic regression.

5.2 Multiclass Classification

To deal with problems with more than two classes, two main options exist:

- One vs the rest: Define K classes $(1, 2, ..., K)$ compare each class to all the others as in a two-class problem:

$$\begin{aligned} i &\rightarrow y_\mu = +1\,, \\ (1, 2, ..., i-1, i+1, ..., K) &\rightarrow y_\mu = -1\,. \end{aligned}$$

  This way we are back to a two-class problem.

- Multiclass logistic regression: We consider K classes. Each sample X_μ belong to one of them. We denote $C_\mu = k$ the class sample μ is belonging to. The classes are described with so-called *one-hot-encoding*:

$$y_\mu \in \mathbb{R}^K: \quad y_\mu = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \text{if} \quad C_\mu = 1, \quad y_\mu = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \quad \text{with the 1 in the } k\text{-th row if} \quad C_\mu = k\,, \tag{5.12}$$

  The labels are then collected in a matrix $y \in \mathbb{R}^{n \times K}$ and the considered probabilistic model $P_{\mu a}$ will be the probability that point μ is in class a.

$$a = 1, ..., K \quad p_{\mu a} = P_{\text{out}}\left(y_{\mu a} = 1 | X_\mu, w^* \in \mathbb{R}^{d \times K}\right) = \frac{e^{\sum_{i=1}^d X_{\mu i} w^*_{ia}}}{\sum_{b=1}^K e^{\sum_{i=1}^d X_{\mu i} w^*_{ib}}}\,. \tag{5.13}$$

  The posterior then reads:

$$P(w|X, y) = \frac{1}{Z} P_w(w) \prod_{\mu=1}^n \left[\prod_{a=1}^K p_{\mu a}^{y_{\mu a}} \right]\,. \tag{5.14}$$

  This makes sense because if for example $C_\mu = b$, $\prod_{a=1}^K p_{\mu a}^{y_{\mu a}} = p_{\mu b}$ as $y_{\mu b} = 1$ and $y_{\mu a} = 0$ for any $a \neq b$, and that is what we want for the likelihood. From here, we want to maximize the likelihood or equivalently minimize the loss:

$$\hat{w} = \text{argmin}\, \mathscr{L}(w) \tag{5.15}$$

$$\text{with} \qquad \mathscr{L}(w) = -\frac{1}{n}\sum_{\mu=1}^{n}\sum_{a=1}^{K} y_{\mu a}\log(p_{\mu a}) = -\frac{1}{n}\sum_{\mu=1}^{n}\sum_{a=1}^{K} y_{\mu a}\log\left(\frac{e^{X_\mu\cdot w_a}}{\sum_b e^{X_\mu\cdot w_b}}\right) \tag{5.16}$$

which is called the cross-entropy loss. Once again, one usually adds a regularization. The minimization over w leads to an optimal $\hat{w}$. We then calculate $p_{\text{new,a}}$ and the prediction is $y_{\text{new}} = \text{argmax}_a\, p_{\text{new,a}}$.

It is illustrative to see how to get back the 2-class logistic regression. The multiclass cross-entropy reads:

$$y_\mu \in \{0,1\}; \quad \mathscr{L}(\omega) = -\sum_{\mu=1}^{n}[y_\mu \ln p_\mu + (1-y_\mu)\ln(1-p_\mu)], \tag{5.17}$$

where:

$$\begin{aligned} p_\mu &= \frac{e^{X_\mu\cdot w_1}}{e^{X_\mu\cdot w_1}+e^{X_\mu\cdot w_2}} = \frac{1}{1+e^{X_\mu\cdot(w_2-w_1)}} = P(y_\mu=1), \\ 1-p_\mu &= \frac{1}{1+e^{-X_\mu\cdot(w_2-w_1)}} = P(y_\mu=0). \end{aligned} \tag{5.18}$$

Call $w = -w_2 + w_1$, then:

$$\begin{aligned} &\text{for } y_\mu = 1: \quad \mathscr{L}(\omega) = -\sum_{\mu=1}^{n}\ln\frac{1}{1+e^{-X_\mu\cdot w}}, \\ &\text{for } y_\mu = 0: \quad \mathscr{L}(\omega) = -\sum_{\mu=1}^{n}\ln\frac{1}{1+e^{+X_\mu\cdot w}}. \end{aligned} \tag{5.19}$$

Going back to labels ± 1:

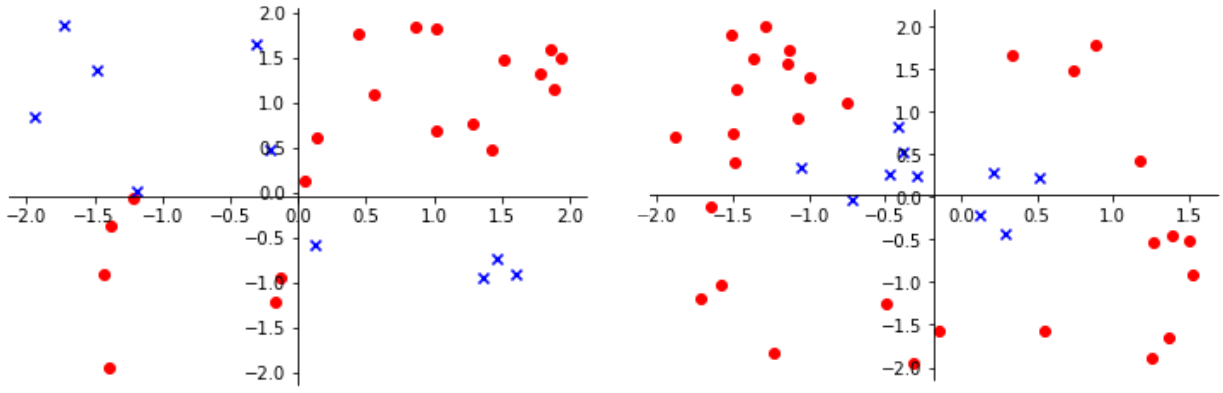
- $1 \leftrightarrow 1$
- $0 \leftrightarrow -1$

we obtain back the logistic loss:

$$\mathscr{L}(w) = \frac{1}{n}\sum_{\mu=1}^{n}\ln\left(1+e^{-y_\mu X_\mu\cdot w}\right) \tag{5.20}$$

as before up to a factor 2 (absorbed in w).

5.3 How to classify data that are not linearly separable?



1. The first way to solve this problem would be to make the data separable as for example by adding an extra dimension or changing coordinates. We will come to this in details in Lecture 10.

2. A simpler way would be to use the k-nearest neighbours method.

$$\hat{y}_{\text{new}}(X_{\text{new}}) = \frac{1}{k} \sum_{\mu: X_\mu \in N_k(X_{\text{new}})} y_\mu \tag{5.21}$$

with $N_k(X_{\text{new}})$ the set with the k closest points to X_{new} from the training set.

In the case of classification (two classes of +1 and -1), one has:

$$\hat{y}_{\text{new}}(X_{\text{new}}) = \text{sign}\left(\sum_{\mu, X_\mu \in N_k(X_{\text{new}})} y_\mu \right). \tag{5.22}$$

The choice of k is made from cross-validation.

However, neighbourhood-based methods in high dimensions face the **curse of Dimensionality**: when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. The curse of dimensionality refers to the problem of finding structure in data embedded in a highly dimensional space. In a high-dimensional space, most points (taken from a uniformly random finite set of N data points inside a finite volume) are far away from each other, and the more features (dimensions) we have, the more data points we need in order to fill the space.

Example: Geometry of the hypercube in high dimensions

Let us take a cube in d dimensions of side 1, so that its volume is 1. Then, let us take a sub-cube of size $0 < l < 1$. Its volume is l^d. If one wants to cover $r = 10\%$ of the volume of the bigger cube. One would actually need:

$$l^d = 0.1 = r \Rightarrow l = r^{\frac{1}{d}} \tag{5.23}$$

So, if $d = 2$, $l = 0.31$, if $d = 10$, $l = 0.79$, and if $d = 100$, $l = 0.98$. So, if $d = 100$, one needs a cube with a side of size 98% of the size of the original cube to cover 10% of its volume. Most of the volume is far away.

Furthermore, take the same hypercube with side length equal to 1 and volume 1 in a d-dimensional space. We want to cover this volume with N smaller cubes homogeneously distributed in the bigger d-dimensional hypercube, each containing a data point. Each of these small cubes will have a volume equal to $\frac{1}{N}$. Their side length l would be:

$$l = \left(\frac{1}{N}\right)^{\frac{1}{d}} \tag{5.24}$$

For a finite N, l converges to 1 when d goes to infinity. Namely, the small cubes have "almost" the same volume as the bigger cube. In an infinite dimensional space, you can put N cubes of volume 1 inside a cube of volume 1. So, if one wants to increase d, one needs a number of points that grows exponentially with d to cover the volume.

Then, how to deal with large dimensions d?

(a) Separate linearly (Lecture 10).

(b) Reduce the dimension (Unsupervised Learning, Lecture 6).

6 Lecture 6: Unsupervised learning and dimensionality reduction 29.10.

Students designated to write this section: Marchand Charles Edouard Eliott Nils, Malashin Ivan, Malanyuk Oleg, Maier Aude, Maier Antoine, Linteau David
Coordinator: Maier Antoine
TA: Federica (TA approved)
Students who actually contributed: Marchand Charles Edouard Eliott Nils, Malanyuk Oleg, Maier Aude, Maier Antoine, Linteau David

In the previous chapters, we focused on Supervised Learning, this means that we learned methods that can be applied when we are in presence of labelled data. In this case, the methods will learn from the training input-output pairs by inferring a function between the data and the labels. We can then apply this function on new data to

predict their label. Now we will be interested in Unsupervised Learning, i.e., algorithms where the data do not have pre-assigned labels. In this case, we cannot use the same methods as before because there is no set of labels to assess if the output of the model is correct or wrong. Instead, the algorithm will discover "itself" patterns in the data.

In this chapter, the focus will lay on a common Unsupervised Learning algorithm: *Principal Component Analysis* (PCA). The idea of PCA is to apply an orthogonal linear transformation to the data such that the first coordinate in the new space is the direction along which the input data varies the most (first greatest variance), the second coordinate the second greatest variance, etc. This method can be used to reduce the dimensionality of the data in such a way that the data structure is preserved (and hence sometimes resolve the curse of the dimensionality we face in high dimension).

6.1 Motivating Examples

1. Mapping human genome

   Novembre, Johnson, Bryc, *et al.* published a paper about the link between the geographic location of people and their genetic information [14]. They choose 3129 people in Europe and sequence their genome. They encoded this data using 500'568 loci (specific locations on a chromosome) using SNP chips (Single Nucleotide Polymorphism). After having cleaned the data, which means removing people with uncertain ancestry or mixed/non-European origin and keeping only the loci with high quality and variability, they end up with 1387 people and 197146 loci. By applying the Principal Component Analysis (PCA) on the matrix $X \in \mathbb{R}^{1387\times 197146}$, they found the following graphic:

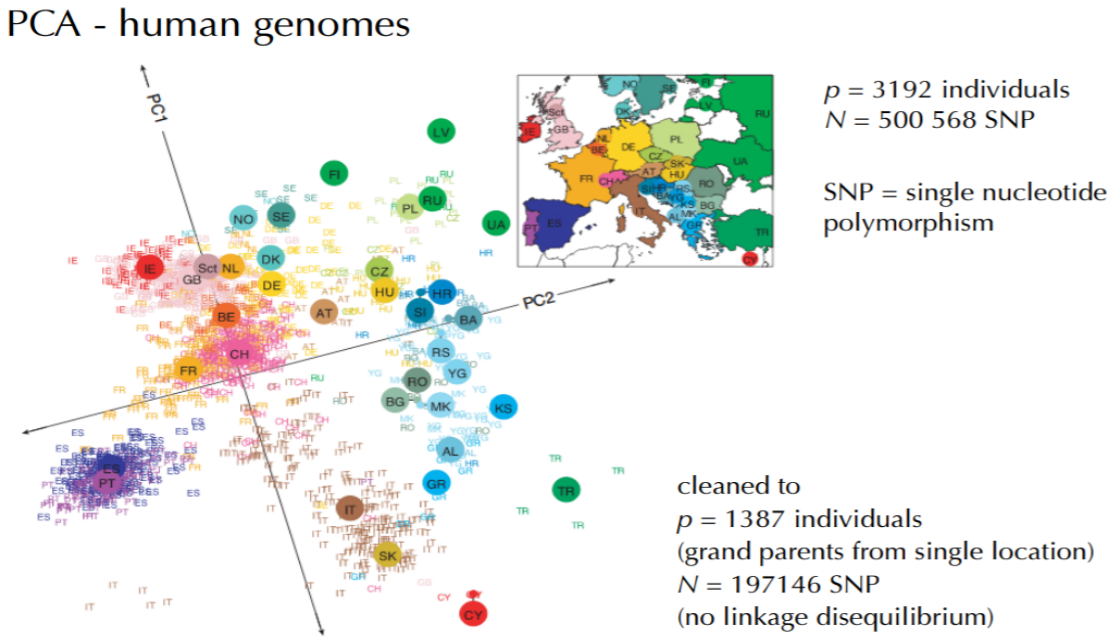


Figure 15: Result of the study [14] showing the correlation between geographic and genetic information of people

   The two axes of this graphic are the two principal components detected trough PCA, associated to the two most relevant singular values. The coloured circles represent the geographic position of different European states. The individual points, whose position is obtained with respect to the two principal components (PC1,PC2), are coloured according to the location of the grandparents of the subject and denoted by the first letters of the country. One observes that there is a high correlation between the genome of the people and the map of Europe, which means that the genome contains in some sense the map of Europe.

2. Movie recommendations

   In order to recommend a movie, companies such as Netflix used to ask their clients to rate the movies they watched and then build a big matrix X where the rows represent the users, the columns the movies available and the element X_{ij} is the rating of user i for the film j (from 1 to 5 for example). The matrix $X \in \mathbb{R}^{n\times d}$,

where n is the number of users and d the number of movies, will only be partially filled and might in a simplified version like:

$$X = \begin{array}{|c|c|c|c|c|} \hline & 1 & 5 & 4 & \\ \hline 3 & & 2 & & \\ \hline & 1 & & 4 & \\ \hline & & 5 & 3 & \\ \hline 2 & & 3 & & \\ \hline \end{array} . \tag{6.1}$$

With PCA one can predict if one movie will be appreciated by a specific user with a certain accuracy.

6.2 Singular value decomposition (SVD)

Let $X \in \mathbb{R}^{n\times d}$ be a matrix (usually of data) that is non-symmetric. For our purposes, X will be a matrix of data. The *singular value decomposition* of X consists of writing this matrix as follows:

$$X = U\Sigma V^{\mathrm{T}}, \tag{6.2}$$

where $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{d\times d}$ are real orthonormal matrices ($UU^T = \mathbb{I}_n, VV^T = \mathbb{I}_d$). The matrix $\Sigma \in \mathbb{R}^{n\times d}$ is diagonal with entries $\Sigma_{ii} = \sigma_i$ called the singular values of X. The singular values are non-negative and real. We call the columns of U *left singular vectors*, while the rows of V^T are called *right singular vectors*. Including the dimension of each matrix, equation (6.2) reads

$$\underset{n\times d}{X} = \underset{n\times n}{U} \times \underset{n\times d}{\Sigma} \times \underset{d\times d}{V^T}. \tag{6.3}$$

The rank of X (i.e., the number of linearly independent columns or rows) is equal to the number of singular values greater than 0 ($\sigma_i > 0$). If r denotes the rank of X, then r must satisfy $r \leq \min(n, d)$.

In order to compute the SVD we use a procedure based on the eigendecomposition of the covariance matrix X^TX (or XX^T). This is a standard way to proceed for PCA. We start with a matrix of data X and perform SVD so that $X = U\Sigma V^T$, where Σ is a diagonal matrix of singular values σ_i. Then one can rewrite the covariance matrix as follows:

$$\begin{aligned} X^TX &= V\Sigma U^T U\Sigma V^T \\ &= V\Sigma^2 V^T \\ &\equiv V\Lambda V^T, \end{aligned} \tag{6.4}$$

where in the last equality we defined the diagonal matrix Λ with eigenvalues $\lambda_i = \sigma_i^2$. From this, we can deduce that the eigenvectors of X^TX andXX^T are then the right and left singular vectors. These in turn correspond to the principal components/directions of X. In the following, we will see how one can reduce the dimensionality of the data using the principal components associated with the largest singular values.

There are 2 important properties of the Singular Value Decomposition:

1. Dimensionality of data subspace

   Let r be the rank of the non-symmetric matrix $X \in \mathbb{R}^{n\times d}$. As the rank refers to the number of linearly independent rows/columns, it cannot be larger than the lowest dimension of the matrix: rank $r \leq \min(n, d)$. To see this, one can rotate the matrix X by V:

   $$XV = U\Sigma V^TV = U\Sigma \tag{6.5}$$

   where Σ is a diagonal matrix with only r non-zero components, hence the resulting matrix $U\Sigma$ will only have r non-zero columns. As the rotation leaves the original subspace of the data unchanged, this means that the data lies in r dimensional subspace of $\mathbb{R}^d$. In the Figure 16 below, one can see the principle illustrated for a matrix of rank 3.

2. Young-Eckart theorem

   For a real non-symmetric matrix $X \in \mathbb{R}^{n\times d}$, let $X^{(k)}$ be a rank-k approximation of the matrix X. The best approximation that would minimize the Frobenius norm:

   $$\|X - X^{(k)}\| = \sum_{\mu i} \left(X_{\mu i} - X^{(k)}_{\mu i}\right)^2 . \tag{6.6}$$

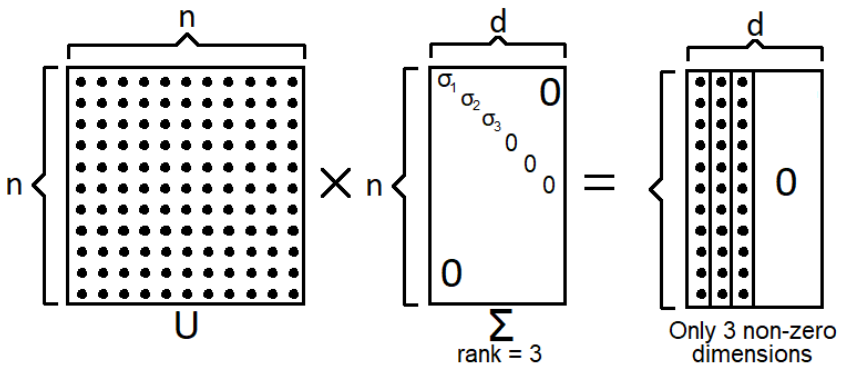


Figure 16: Illustration of the data dimensionality for X of rank 3. After rotation, it is clear that the data is 3D

If we denote the SVD of X by (cf. (6.2)):

$$X_{\mu i} = \sum_{\alpha=1}^{r} U_{\mu\alpha}\sigma_\alpha V_{i\alpha} \tag{6.7}$$

with r the rank of X, then the Young-Eckart theorem states that the best rank-k approximation of X is given by

$$X^{(k)}_{\mu i} = \sum_{\alpha=1}^{k} U_{\mu\alpha}\sigma_\alpha V_{i\alpha}\,, \tag{6.8}$$

where the sum goes over the first k largest values of σ. For proof, see standard linear algebra textbooks. This theorem states that to best approximate the data with only k dimensions (thus reducing its dimension to k) it is best to use the first k singular vector corresponding to the k largest singular values.

6.3 Low-rank matrix completion

Now that we have developed some nice properties of singular values, we will go back to example 2 on movie recommendation and use it to understand how to predict movie rankings by matrix completion.

Intuition: Each person has k descriptors that are relevant (but unknown) for their movie taste, these descriptors are gathered in a matrix $U \in \mathbb{R}^{n\times k}$. For example $U_{\mu 1}$ could be the age, $U_{\mu 2}$ the gender, $U_{\mu 3}$ the income, etc. where $\mu = 1, ..., n$ labels the users. On the other hand, each movie has k descriptors evaluating its accordance with the corresponding users descriptors, these descriptors could be for instance the language of the movie, its genre, the actors playing in it, etc. The only information we have is the sparse matrix $\tilde{X}$ containing the appreciation of some of the movies given by some of the users:

$$\tilde{X} = \underbrace{\left.\begin{array}{|c|c|c|c|c|}\hline & 1 & 5 & 4 & \\\hline 3 & & 2 & & \\\hline & 1 & & 4 & \\\hline & & 5 & 3 & \\\hline 2 & & 3 & & \\\hline\end{array}\right\}}_{d \text{ movies}} n \text{ users} \tag{6.9}$$

The goal is to find the matrices U, $\tilde{V}$ which explain best the data. A method to achieve this goal is to find the matrices U, $\tilde{V}$ such that the following is minimized:

$$\sum_{\mu i}\left(\tilde{X}_{\mu i} - \sum_{\alpha=1}^{k} U_{\mu\alpha}\tilde{V}_{i\alpha}\right)^2 \tag{6.10}$$

which can be done by making use of Young-Eckart theorem. This method goes as follows:
In order to use the Young-Eckart theorem, we need to fill in the empty cells of the matrix $\tilde{X}$. Hence, we first construct

a matrix X such that:

$$\begin{cases} X_{ij} = \tilde{X}_{ij} & \text{for known elements of } \tilde{X} \\ X_{ij} = 0 & \text{for unknown elements} \end{cases} \tag{6.11}$$

Alternative methods fill the unknown elements with row or column averages instead of 0's.
Next, we write the SVD of X as $X_{\mu i} = \sum_{\alpha=1}^{r} U_{\mu\alpha}\sigma_\alpha V_{i\alpha}$. Using Young-Eckart theorem, we know that $\|X - U\tilde{V}^T\|$ is minimized when:

$$\sum_{\alpha=1}^{k} U_{\mu\alpha}\tilde{V}_{i\alpha} = \sum_{\alpha=1}^{k} U_{\mu\alpha}\sigma_\alpha V_{i\alpha} \tag{6.12}$$

where we only took the k leading singular values. Hence we obtain $\tilde{V}_{i\alpha} = \sigma_\alpha V_{i\alpha}$ and the prediction on a new user-movie pair of the ratings is given by:

$$\hat{X}_{\mu i} = \sum_{\alpha=1}^{k} U_{\mu\alpha}\sigma_\alpha V_{i\alpha}. \tag{6.13}$$

6.4 Principal Component Analysis (PCA)

We now explain in more detail the previous example 1, the map of Europe from principal components to illustrate PCA, and have a better understanding of how it works because this example is more suitable for understanding what PCA means, rather than the previous one which was more a direct application of the Young-Eckart theorem.

Principal Component Analysis

Goal: Find the k-dimensional representation of the data X that preserves as closely as possible its structure.

1. Centre and normalize the raw data $\tilde{X}_{\mu i}$ along each direction i as :

$$X_{\mu i} = \frac{\tilde{X}_{\mu i} - \frac{1}{n}\sum_{\tilde{\mu}=1}^{n}\tilde{X}_{\tilde{\mu} i}}{\sqrt{\frac{1}{n}\sum_{\tilde{\mu}=1}^{n}\tilde{X}^2_{\tilde{\mu} i} - \left(\frac{1}{n}\sum_{\tilde{\mu}=1}^{n}\tilde{X}_{\tilde{\mu} i}\right)^2}}, \forall i = 1, ..., d \tag{6.14}$$

   This normalization is introduced because the features of the data could have been measured with different units and scales. This centring and normalizing process yields features that are comparable in the sense that they all have no units and the same scale.

2. Do SVD on $X_{\mu\nu} = \left(U\Sigma V^T\right)_{\mu\nu}$ by computing the eigen-decomposition of the covariance $X^T X$.

   Take the k first right-singular vectors $V \in \mathbb{R}^{d\times k}$.

   Project as: $XV = U\Sigma \in \mathbb{R}^{n\times k}$.

   $\longrightarrow$ Get the map of Europe by plotting n points (each for one person) in the k-dimensional space.

6.4.1 Two remaining Questions

How can we use PCA? The origin of the person was absent from the original dataset. The colours on the Figure 15 have been added after applying an algorithm to separate the dimension reduced dataset in the number of classes corresponding to the number of countries to visualize the origin of the people. Therefore, if we want to know the origin of a new person for which we have its genetic information, we could add it to X and place it on the map (the plane (PC1, PC2)). In order to determine its origin, we could then run local classification methods, such as k-nearest neighbours, on this 2D-space identified with PCA.

How to Choose k in PCA? We have not specified yet how the value of k should be determined. Moreover, we do not have a test set to do cross validation as it was the case in supervised learning. Let us try to motivate here how we can find k.

Recall that singular values of X are eigenvalues of XX^{T} (or $X^{\mathrm{T}}X$). If we compute these eigenvalues and plot them on a histogram, it is often the case that the histogram will have the behaviour seen in Figure 17. It is then easier from such a histogram to postulate an appropriate value for k.

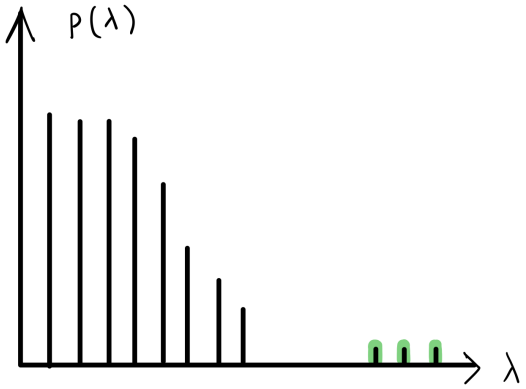


Figure 17: Histogram of the eigenvalues λ of XX^{T} (or $X^{\mathrm{T}}X$) – associated to the singular values σ of X – are shown. Three isolated (single) eigenvalues are highlighted in green. These give a good hint that one should pick $k = 3$. The other vertical lines correspond to a large number of small eigenvalues (which does not indicate any structure in the data).

6.5 Low-rank matrix estimation model

Let us consider a "spin glass card game" example to further illustrate how PCA works. We will use this example in the next lectures to illustrate other concepts and the relation with statistical mechanics. Consider N people, each receiving at random a card with values ± 1. Let $S_i^* \in \{\pm 1\}$ denote the value of the card of person i. For each pair (ij) the statistician observes the quantity Y_{ij} that we construct as:

$$Y_{ij} = \frac{1}{\sqrt{N}} S_i^* S_j^* + W_{ij}, \tag{6.15}$$

where $W_{ij} \sim \mathscr{N}(0, \Delta)$, $\Delta \sim O(1)$, and we further assume that N is large. Given that we only observe Y_{ij}, the goal is to split the N people in two groups such that as many people with the same card are in the same group. This problem can be solved using PCA. It turns out that here $Y_{ij} = Y_{ji}$, that is, $Y \in \mathbb{R}^{N \times N}$ is a symmetric matrix. So here SVD reduces to an eigenvalue decomposition. According to the notation introduced above, and in this special case taking $U = V$, we have:

$$Y_{ij} = \sum_{\alpha,\beta=1}^{N} U_{i\alpha} \Lambda_{\alpha\beta} U_{j\beta} = \sum_{\alpha=1}^{N} \lambda_\alpha U_{i\alpha} U_{j\alpha}, \tag{6.16}$$

where in the second equality we used the fact that $\Lambda_{\alpha\beta} = \lambda_\alpha \delta_{\alpha\beta}$. The spectrum of Y is schematically shown in Figure 18, where two cases $\Delta < 1$ and $\Delta > 1$ are considered.

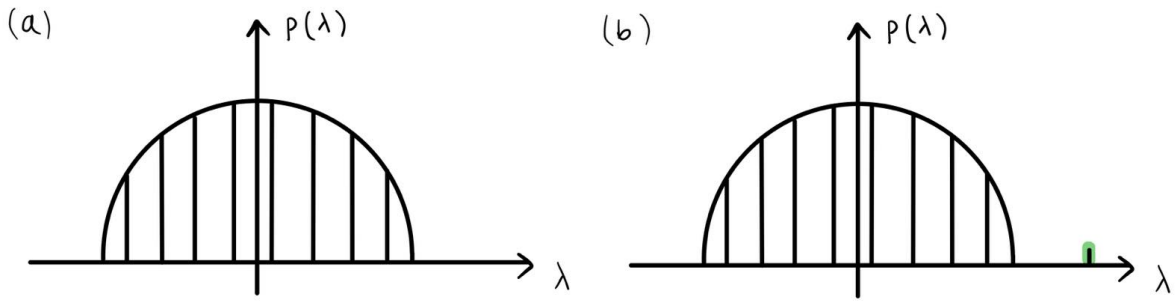


Figure 18: Spectrum of Y, as defined in equation (6.15), for (a) $\Delta > 1$ and (b) $\Delta < 1$. When the noise Δ (or standard deviation of the normal distribution) is smaller than 1, we see that the largest eigenvalue (highlighted in green) "lives" outside of the bulk (the bulk lives under the Wigner semi-circle). On the other hand, when $\Delta > 1$, there is no outlying eigenvalue (no structure found) because the noise Δ is too large.

From Figure 18 (and the above discussion), it is clear that in this example $k = 1$ is a good choice because there is one eigenvalue living outside of the bulk in the regime $\Delta < 1$. When $\Delta < 1$, we can see that the relevant information

in Y is contained in the first principal component associated with $\lambda_{\max}$ (as it is the component with largest variance). However, when $\Delta > 1$, there is no more structure in the data. We can then say that $\Delta = 1$ defines a phase transition.

Let $v^{\max}$ be the eigenvector associated with the largest eigenvalue $\lambda_{\max}$. The components $v_i^{\max}$ of $v^{\max}$ are plotted (schematically) in Figure 19.

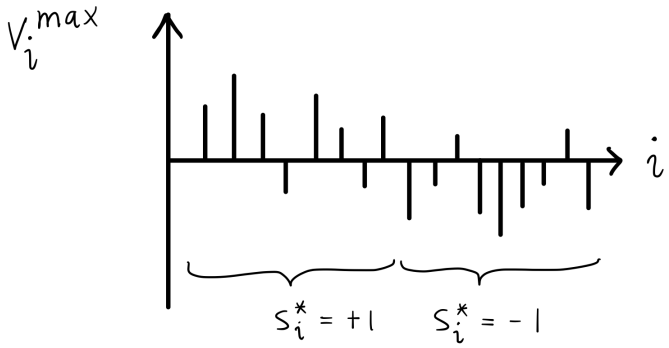


Figure 19: Components of the eigenvector $v^{\max}$ associated with the largest eigenvalue $\lambda_{\max}$, for some $\Delta < 1$.

Let us define the PCA estimator as:

$$\hat{S}_i^{\text{PCA}} = \text{sign}(v_i^{\max}), \tag{6.17}$$

and further define the overlap (between the estimator and the actual value)

$$Q = \frac{1}{N}\Big|\sum_i \hat{S}_i^{\text{PCA}} S_i^*\Big|. \tag{6.18}$$

In Figure 20, we consider Q as a function of Δ. We can clearly see that there are two regimes: $\Delta < 1$ and $\Delta > 1$. When $\Delta > 1$, the quantities $\hat{S}_i^{\text{PCA}} S_i^*$ average out to 0 ($Q = 0$) due to the noise, while if we reduce Δ below the threshold of 1, the quantities $\hat{S}_i^{\text{PCA}} S_i^*$ average out to a non-zero value (to eventually reach 1 if the noise is turned off altogether).

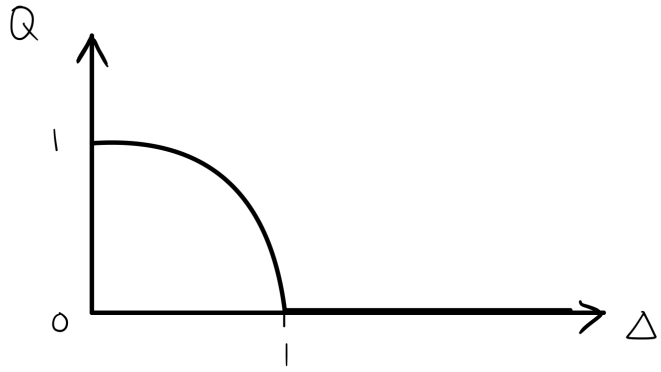


Figure 20: Overlap Q, as defined in equation (6.18), is plotted as a function of the noise (standard deviation Δ of a normal distribution). A phase transition can be seen at $\Delta = 1$.

7 Lecture 7: Inference and Statistical Physics & Monte Carlo Sampling – Part I 5.11.

Students designated to write this section: Koller Cédric Xavier, Kuang Luochen, Leontsinis Matthew James, Leroy Gabin Paul Jacques, Libardi Gabriele, Linder Louis
Coordinator: Leontsinis Matthew James
TA: Giovanni (TA approved)
Students who actually contributed: Koller Cédric Xavier, Kuang Luochen, Leontsinis Matthew James, Libardi Gabriele, Linder Louis

7.1 Reminder of The Spin Glass Card Game

We first recall the main concepts of the *spin glass card game* introduced in the previous lecture. N people receive at random a card with values $S_i^* \in \{\pm 1\}$ where $i \in \{1, ..., N\}$ indicates the person. Each pair of persons (ij) gives the information

$$Y_{ij} = \frac{1}{\sqrt{N}} S_i^* S_j^* + W_{ij}, \; W_{ij} = W_{ji} \sim \mathcal{N}(0, \Delta) \tag{7.1}$$

The goal is to infer S^* (up to the sign by the symmetry of the problem) given the information Y, but naturally not S^* or W.

In Lecture 6, we introduced the PCA estimator $\hat{S}_i^{\text{PCA}} = \text{sign}(v_i^{\max})$, where $v_i^{\max}$ is the eigenvector of Y corresponding to the largest eigenvalue $\lambda_{\max}$. The following result was also introduced. The overlap, defined in (6.18) behaves in the limit $N \to \infty$ as:

$$Q \begin{cases} \overset{N\to\infty}{>} 0 & \text{when } \Delta < 1, \\ \overset{N\to\infty}{=} 0 & \text{when } \Delta > 1. \end{cases} \tag{7.2}$$

The *spin glass card game* is an example of clustering. Indeed, the two groups of people with cards/spins $S_i^* = \pm 1$ form two possible clusters. Other examples of clustering are found in medicine (e.g., clustering of patients by treatment types), in quantitative biology, or in social networks (e.g. clustering friends or enemies). Clustering will be further discussed in lecture 9.

Two natural questions arise from the spin glass card game:

1. Why the name *spin glass*?
2. Is there a better estimator than PCA?

In what follows we will answer these questions starting with the second one that is related to the topic of optimal inference. We have, so far, seen the following two estimators:

- **Maximum likelihood estimator** (ML)
- **Maximum a posteriori estimator** (MAP)

In the following we will introduce the **Bayesian-Optimal estimation** and the **Minimum Mean Square Error** (MMSE) estimator.

7.2 Bayesian Inference and the MMSE estimator

The posterior probability distribution is, from Bayes rule (cf. Eq.(2.11)),

$$P(S|Y) = \frac{P_{\text{out}}(Y|S) P_S(S)}{\tilde{Z}}, \tag{7.3}$$

where the quantity $\tilde{Z}$ normalises the posterior distribution and is given by:

$$\tilde{Z} = P(Y) = \sum_{S_1,\dots,S_N} P(Y, S) = \sum_{S_1,\dots,S_N} P_{\text{out}}(Y|S) P_S(S), \tag{7.4}$$

where the sum is over all possible configurations of spins (i.e., we are summing the joint probabilities of Y with every possible outcome of S). This concept was covered in lecture 2; for a refresher please refer to Eqs.(2.7) and (2.9). Since every spin is either $+1$ or -1, the distribution for the spin vector is given by the Rademacher probability distribution:

$$P_S(S) = \frac{1}{2^N} \prod_{i=1}^{N} (\delta_{S_i,-1} + \delta_{S_i,1}). \tag{7.5}$$

Since we added a Gaussian noise with variance Δ to $(S_i S_j) N^{-1/2}$ to get Y_{ij}, the likelihood is Gaussian:

$$P_{\text{out}}(Y|S) = \prod_{i<j} \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}\left(\frac{S_i S_j}{\sqrt{N}} - Y_{ij}\right)^2}. \tag{7.6}$$

Plugging those equations in (7.3), we get the posterior:

$$P(S|Y) = \frac{1}{\tilde{Z}2^N}\prod_{i=1}^{N}(\delta_{S_i,1}+\delta_{S_i,-1})\prod_{i<j}\frac{1}{\sqrt{2\pi\Delta}}e^{-\frac{1}{2\Delta}\left(\frac{S_iS_j}{\sqrt{N}}-Y_{ij}\right)^2} \tag{7.7}$$

Hiding the S-independent terms in a new normalization[11], Z, we get for the posterior probability distribution:

$$\underbrace{P(S|Y)}_{\text{Posterior}} = \underbrace{\frac{1}{Z}}_{\text{Evidence}}\underbrace{\prod_{i=1}^{N}(\delta_{S_i,1}+\delta_{S_i,-1})}_{\text{Prior}}\underbrace{\prod_{i<j}e^{\frac{1}{\Delta\sqrt{N}}S_iS_jY_{ij}}}_{\text{Likelihood}}. \tag{7.8}$$

7.2.1 Analogies with Statistical Physics

Recalling the formalism of statistical physics, we see that the posterior probability distribution takes the form of a Boltzmann distribution. The normalization Z takes the place of the partition function. The noise variance Δ plays the role of the temperature and the corresponding Hamiltonian is:

$$\mathscr{H}(S) = \frac{1}{\sqrt{N}}\sum_{i<j}Y_{ij}S_iS_j \tag{7.9}$$

which reminds us of the Ising model, with Ising spin variables. Note, however, the difference that the interactions Y_{ij} are different for every pair of spins. An Ising model with such interactions is called a spin glass, because of its dynamical behaviour that is more similar to a glassy material than to an ordinary magnet or liquid.

The Table 1 summarizes the Bayesian-inference and Statistical Physics equivalencies. This analogy will help us interpret and understand some of the properties and behaviour of statistical inference and Machine Learning problems as we can view them through the lens of statistical physics.

Physics	**Machine Learning**
Ising spins S_i	Rademacher variables S_i
Partition function Z	evidence Z
Hamiltonian H	negative log-likelihood
Temperature T, β^{-1}	noise variance Δ

Table 1: Analogy between Physics and Machine Learning when looking at the spin glass card game.

7.2.2 Maximum Likelihood: PCA

It is illustrative to restate the **maximum likelihood estimator** for the spin glass card game problem (or low-rank matrix estimation more in general). From (7.8) the negative log-likelihood is proportional to:

$$\sum_{i<j}\left(\frac{S_iS_j}{\sqrt{N}}-Y_{ij}\right)^2. \tag{7.10}$$

This is the same quantity that the principal component analysis with $k=1$ minimizes. PCA finds the vector S that minimises the negative log-likelihood. We saw that the solution is

$$\hat{S}_i^{\text{ML}} = v_i^{\max}, \tag{7.11}$$

where $v^{\max}$ is the leading eigen-vector.[12] In the principal component analysis and the maximum likelihood, we ignore the fact that $S_i = \pm 1$. We also drop the sign function with respect to (6.17), both options are considered in practice depending on the application.

[11] For completeness note that the bracket in the exponent of equation (7.7) is multiplied out and then simplified by taking advantage of the fact that $S_i \in \{-1,1\}$, $Y_{ij} \in \{-1,1\} \forall i,j$ and that the squares of these are always equal to one.

[12] Note as well that here we take into account the prior.

7.2.3 Maximum A Posteriori Estimator: MAP

We also illustrate what is the **maximum a posteriori estimator** in the language of the statistical physics analogy. In MAP we wish to find the spins S that maximise the posterior:

$$\max_S P(S|Y) = \max_S P(S)P(Y|S), \tag{7.12}$$

which is achieved by minimising the Hamiltonian:

$$\mathscr{H}(S) = -\frac{1}{\sqrt{N}} \sum_{i<j} Y_{ij} S_i S_j, \tag{7.13}$$

where spins are $S_i \in \{-1, 1\}$ such that (7.13) is consistent with the definition of the prior given earlier. This is equivalent to finding the ground state of a Hamiltonian with Ising spins. We note that finding ground states of such disordered systems is generally computationally hard, in practice and without a guarantee of success the simulated annealing introduced in Lecture 8 can be used.

7.2.4 Minimum Mean Square Error Estimator

The **Minimum Mean Square Error** (MMSE) estimator seeks to minimise the mean square error (MSE), which is defined as the average over the components of the square of the residuals. In the Bayesian approach, where some prior information is known about the parameter we aim to estimate, the MSE (a quadratic cost function) is minimised. That is, we aim to find $\hat{S}(Y)$ such that:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} \left(S_i^* - \hat{S}_i(Y) \right)^2 \tag{7.14}$$

is minimized, where $\hat{S}$ is the estimator function of the measurement, Y, and the S_i^* are the ground truth that we do not know (i.e., they are hidden variables). There is an issue: Possessing no knowledge of S^*, we cannot evaluate (7.14) directly. We can, however, use the fact that S^* is as if it was drawn from the posterior. That is, $S^* \sim P(S|Y)$, so that we may now seek to find the MSE averaged over the posterior for the estimator $\hat{S}(Y)$:

$$\min_{\hat{S}(Y)} \sum_{\{S_i\}_{i=1}^N} P(S|Y) \frac{1}{N} \sum_{i=1}^{N} \left(S_i - \hat{S}_i(Y) \right)^2. \tag{7.15}$$

Taking the partial derivative with respect to $\hat{S}_j$ and setting equal to 0, we have:

$$\sum_{\{S_i\}_{i=1}^N} P(S|Y)(S_j - \hat{S}_j(Y)) = 0 \Rightarrow \sum_{\{S_i\}_{i=1}^N} P(S|Y) S_j = \hat{S}_j(Y), \tag{7.16}$$

note that the sum of the posterior over the S_j's is equal to unity as the posterior is normalised. Hence in order to estimate the best value of S_j that minimises the MSE, we must average the S_j w.r.t. the posterior probability distribution. There is no need to compute the maximum of the distribution, contrary to the MAP approach in Section 7.2.3, instead the average over the posterior need to be computed.

In order to understand the implications of the MMSE, we first examine the posterior. Recall that this is nothing else than the Boltzmann measure as noted in section 7.2.1; that is $\frac{1}{Z} e^{-\beta H}$. We are computing the average of some spin (S_j), hence the quantity in (7.16) is in fact the local magnetisation, m_j, in the corresponding Ising spin glass model. Additionally, we may define the quantity $\mu(S_j)$ as:

$$\mu(S_j) = \sum_{\{S_i\}_{i=1}^N \setminus S_j} P(S|Y), \tag{7.17}$$

often referred to as the *marginal probability distribution*. Finally, (7.16) may now be written more compactly as:[13]

$$\hat{S}_i(Y) = \sum_{S_i} S_i \mu(S_i). \tag{7.18}$$

The question is now how to compute the magnetisation efficiently. This will be covered in the next part.

[13] Note that this posterior averaging can be seen as an average over all configurations which are as likely as the ground truth.

7.3 Monte Carlo Sampling

7.3.1 Markov Chains

A *Markov Chain* (MC) or *Markov process* is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. A Markov Chain is defined by a sequence of random variables $\{S_t\}_{t=1}^{N}$. In order to link these random variables, a transition matrix $T(\vec{S} \to \vec{S}') \in \mathbb{R}^{2^N \times 2^N}$ is introduced, which is the probability for the transition $S \to S'$ from time step t to $t+1$. In particular, this concept may be expressed mathematically through the formula for a conditional probability:

$$T(\vec{S} \to \vec{S}') = P(\vec{S}^{(t+1)} = \vec{S}' | S^{(t)} = \vec{S}). \tag{7.19}$$

If we initialize the Markov Chain in one state S^0, then $P^0(S) = \delta_{S,S^0}$. We can then evolve P^0 and compute the probability that at time t the chain will be in state S, by evolving P^0 for t steps using (7.20). Consider that at time t we have the probability distribution $P^t(\vec{S})$; at time $t+1$ we compute the probability distribution P^{t+1}, thanks to the following relation:

$$P^{t+1}(\vec{S}') = \sum_{\{S_i\}_{i=1}^{N}} T(\vec{S} \to \vec{S}') P^t(\vec{S}), \tag{7.20}$$

which depends on the distribution from the previous time step. We will be interested in Markov chains that converge to a stationary/equilibrium distribution. Denoting such an equilibrium probability distribution $P_{\text{eq}}(\vec{S})$ for $t \to \infty$, we can write:

$$P_{\text{eq}}(\vec{S}') = \sum_{\vec{S}} T(\vec{S} \to \vec{S}') P_{\text{eq}}(\vec{S}). \tag{7.21}$$

A very widely used condition for convergence of a Markov chain to equilibrium is called the *Detailed Balance Condition*. The transition matrix T satisfies the Detailed Balance Condition with respect to the distribution $\tilde{P}$ if:

$$\tilde{P}(\vec{S}) T(\vec{S} \to \vec{S}') = \tilde{P}(\vec{S}') T(\vec{S}' \to \vec{S}). \tag{7.22}$$

If the Detailed Balance Condition is satisfied (and the Markov Chain is ergodic[14], though this condition is generally less restrictive), the *Monte Carlo Markov Chain* (MCMC) converges to $\tilde{P}$, i.e., $P_{\text{eq}} = \tilde{P}$ for finite N and infinite t. In general, the Detailed Balance Condition is a sufficient condition for convergence to P_{eq}, but not a necessary one.

7.3.2 The Metropolis-Hastings Algorithm

Monte Carlo methods based on Markov chains (MCMC) are a class of algorithms for sampling from probability distributions based on the construction of a Markov chain, having the desired distribution as an equilibrium (or stationary) distribution. In general, we have a probability that we wish to sample from, then design a Markov-chain-based algorithm to sample from it. After simulating a large number of steps in the chain, the extracted values can be used as a sample of the desired distribution.

We will now introduce an example of Monte Carlo Markov Chain: The Metropolis-Hastings Algorithm. The Metropolis-Hastings algorithm is an MCMC method used to generate values $x_1, x_2, \ldots, x_n$, which have some probability distribution, $p(x)$. The function $p(x)$ does not need to be known; it is sufficient that a function $f(x)$ proportional to $p(x)$ is known. This weak requirement allows us to use the Metropolis-Hastings algorithm in Bayesian statistics to sample from posterior distributions whose integral is too difficult, or impossible, to compute in the analytic form.

Let us start with an example that one may be familiar with from statistical physics. Considering the specific case of the Ising spin glass model with Hamiltonian $\mathscr{H}(S) = -\frac{1}{\sqrt{N}} \sum_{i<j} Y_{ij} S_i S_j$. We aim to sample the Boltzmann probability density function:

$$P(S) = \frac{e^{-\mathscr{H}(S)/\Delta}}{Z}. \tag{7.23}$$

Given Y_{ij}, Δ, N we can define the algorithm in the following steps:

1. Start with a random configuration $\{S_i\}_{i=1}^{N}$ where $S_i = \pm 1$;
2. Pick a element j uniformly in $\{1, \ldots, N\}$, compute the local field felt by spin j, $H_j = \sum_{k \neq j} S_k Y_{kj}$ and:
   - if $S_j H_j \geq 0$ (that means that the considered spin is aligned to the local field) then flip $S_j \to -S_j$ with a probability of $p = e^{-\frac{2 S_j H_j}{\Delta}}$,

[14] Without giving a complete definition, an ergodic system is a dynamic system which eventually explores the entire space in a random uniform way.

- if $S_j H_j < 0$ (that means that the considered spin's direction is opposite to the local field's) then flip $S_j \to -S_j$;

3. Repeat point 2 for T steps. After the equilibration time (T_{eq}), return S every T_{decorr} steps.

The equilibration time T_{eq} is the period of time that elapses between the begin of the simulation and the achievement of a condition of stability of the system, in particular this manifests itself with the convergence of the quantities measured to a given plateau.

The decorrelation time T_{decorr} is the time one must wait in order for the configurations S^t and $S^{t+T_{\text{decorr}}}$ to be almost independent. Since it is hard to check for independence we content ourselves with checking the lack of correlations (independent implies uncorrelated but the converse is false).

After introducing this specific example of the Metropolis-Hasting algorithm as a sampling method for the Ising model, we will elaborate further on this important algorithm more in general.

Generally speaking, the Metropolis-Hastings algorithm is used to obtain the random samples from a probability distribution when direct sampling is difficult. When the Markov Chain is being processed from some initial state $\vec{S}$ to the next state $\vec{S''}$, a random trial state $\vec{S'}$ will be generated as the candidate for the next state $\vec{S''}$. The Metropolis-Hastings algorithm will then accept all $\vec{S'}$ as $\vec{S''}$ if $P_{\text{eq}}(\vec{S}) \leq P_{\text{eq}}(\vec{S'})$. On the contrary, if $P_{\text{eq}}(\vec{S}) > P_{\text{eq}}(\vec{S'})$, the probability to accept $\vec{S'}$ as being $\vec{S''}$ will be $p = \frac{P_{\text{eq}}(\vec{S'})}{P_{\text{eq}}(\vec{S})}$. In other words, we have:

$$p = \min(1, \frac{P_{\text{eq}}(\vec{S'})}{P_{\text{eq}}(\vec{S})}) \tag{7.24}$$

for the probability to accept the trial state $\vec{S'}$ as our next state in the chain. Repeating this process over a long period will eventually bring the chain to the equilibrium distribution.

This rule of choosing the next state satisfies the detailed balance condition. This is not difficult to prove. For the first case in which $P_{\text{eq}}(\vec{S}) \leq P_{\text{eq}}(\vec{S'})$, we have $p = \min(1, \frac{P_{\text{eq}}(\vec{S'})}{P_{\text{eq}}(\vec{S})}) = 1$ for $\vec{S} \to \vec{S'}$, therefore $T(\vec{S} \to \vec{S'}) = 1$ while $T(\vec{S'} \to \vec{S}) = \frac{P_{\text{eq}}(\vec{S})}{P_{\text{eq}}(\vec{S'})}$. Plugging these into the two sides of the equation for detailed balance (7.22), we find that:

$$\text{l.h.s} = P_{\text{eq}}(\vec{S})T(\vec{S} \to \vec{S'}) = P_{\text{eq}}(\vec{S}) = \frac{P_{\text{eq}}(\vec{S})}{P_{\text{eq}}(\vec{S'})} P_{\text{eq}}(\vec{S'}) = P_{\text{eq}}(\vec{S'})T(\vec{S'} \to \vec{S}) = \text{r.h.s.} \tag{7.25}$$

For the other case, vice versa.

As for the special case of Ising-like models that we have already discussed, we can check our general rule of the Metropolis-Hastings algorithm. For the equilibrium distribution we have:

$$P_{\text{eq}}(\vec{S}) = \frac{e^{-\mathscr{H}(\vec{S})/\Delta}}{Z}, \tag{7.26}$$

where $\mathscr{H}$ is defined as:

$$\mathscr{H} = -\frac{1}{\sqrt{N}} \sum_{i<j} Y_{ij} S_i S_j. \tag{7.27}$$

Then the probability of accepting the flip $S_j \to -S_j$ and $S_i \to S_i$ with all $i \neq j$ will be:

$$p = \min\left(1, \frac{P_{\text{eq}}(-\vec{S}_j)}{P_{\text{eq}}(\vec{S}_j)}\right) = \min\left(1, e^{-\frac{2S_j H_j}{\Delta}}\right). \tag{7.28}$$

8 Lecture 8: Inference and Statistical Physics & Monte Carlo Sampling – Part II 12.11.

Students designated to write this section: Kashko Pavlo, Karlsson Hannes Georg, Kaloyannis Panagiotis Stilianos, Jeandupeux Zoé, Jaubert Timothée Benjamin Antoine, Hu Jinxin
Coordinator: Karlsson Hannes Georg / Kaloyannis Panagiotis Stilianos
TA: Alessandro (TA approved)
Students who actually contributed: Kashko Pavlo, Karlsson Hannes Georg, Kaloyannis Panagiotis Stilianos, Jeandupeux Zoé, Jaubert Timothée Benjamin Antoine

8.1 Monte Carlo Markov Chains

Recap of Metropolis-Hastings

The algorithm was introduced in detail in the previous Lecture 7 (7.3.2). Let us remind the general procedure shortly. Consider a generic probability distribution $P(\vec{S})$ from which we want to sample. The algorithm works as follows:

1. Initiate $\{S_i\}_{i=1}^N$ randomly in $S_i \in \{\pm 1\}$.
2. Let $\vec{S}$ be the current configuration and $\vec{S}'$ the trial configuration.
3. If $P(\vec{S}) \leq P(\vec{S}')$, then we accept the trial configuration $P(\vec{S}')$ with probability $p = 1$.
   If $P(\vec{S}) > P(\vec{S}')$ then we accept the trial configuration $P(\vec{S}')$ with probability $p = \frac{P(\vec{S}')}{P(\vec{S})}$.

In statistical physics, a standard way to choose the trial configuration is to pick at random one spin of the current configuration and to flip it with a certain probability. The Metropolis–Hastings algorithm fulfils the *Detailed Balance Condition*, and will as such converge at large time to the equilibrium distribution. However, this time can be exponentially large.

8.1.1 MCMC for the Biased Card Game

Previously, we assumed that we have had roughly the same amount of $+1$ and -1 cards in the deck. Therefore, there was no bias between the spins, and the magnetization (average card number) of the ground truth was zero. Now, we deviate from this assumption and consider a deck of cards with a fraction ρ of the cards of value $S_i^* = +1$ and a fraction $(1-\rho)$ of value $S_i^* = -1$ with $\rho \in [0,1]$. The rest of the game remains unchanged and the applied algorithm is described in section 7.1. Again, we aim to compute the posterior measure $P(S|Y)$. Comparing with the final expression of the posterior (7.8), the likelihood does not change, only the prior $P(S)$ changes.

$$P(S|Y) = \frac{1}{\tilde{Z}} P(S)P(Y|S) = \frac{1}{\tilde{Z}} \prod_{i=1}^{N} [\rho \delta_{S_i,1} + (1-\rho)\delta_{S_i,-1}] \prod_{i<j} \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}\left(\frac{S_i S_j}{\sqrt{N}} - Y_{ij}\right)^2}. \tag{8.1}$$

By a convenient substitution[15] $\rho = \frac{e^h}{e^{-h}+e^h}$ and hiding S-independent terms in the overall partition function Z,[16] we can rewrite the posterior in terms of h as follows,

$$P(S|Y) = \frac{1}{Z} \prod_{i=1}^{N} e^{S_i h} \prod_{i<j} e^{\frac{1}{\Delta\sqrt{N}} S_i S_j Y_{ij}} = \frac{1}{Z} e^{-\mathscr{H}}, \tag{8.3}$$

with $\mathscr{H} = -\sum_i h S_i - \frac{1}{\Delta\sqrt{N}} \sum_{i<j} Y_{ij} S_i S_j$. Hence, the prior corresponds to the presence of a magnetic field h!

Now, how does the Metropolis update rule changes when there is a magnetic field? The measure towards which we want to converge is the posterior (8.3). Start with a current and a trial configuration differing only by a random spin S_j such that $S_i' = S_i \quad \forall i \neq j$ and $S_j' = -S_j$. Evaluate $\hat{p}$:

$$\hat{p} = \frac{P(\vec{S}'|Y)}{P(\vec{S}|Y)} = \frac{e^{-\frac{1}{\Delta\sqrt{N}} S_j \sum_k S_k Y_{kj}} e^{-hS_j}}{e^{+\frac{1}{\Delta\sqrt{N}} S_j \sum_k S_k Y_{kj}} e^{+hS_j}}. \tag{8.4}$$

The $-$ sign in the numerator comes from the fact that $S_j' = -S_j$. All other terms $i \neq j$ cancel out, giving

$$\hat{p} = e^{-2(\frac{1}{\Delta} H_j + h) S_j}, \text{ with } H_j = \sum_k S_k Y_{kj} \frac{1}{\sqrt{N}}. \tag{8.5}$$

H_j plays the role of the magnetic field felt by the spin j generated by all the other spins, while h is the magnetic field due to the prior.

[15] Note that thanks to this substitution one can write the prior as:

$$\begin{aligned} P(S) &= \prod_{i=1}^{N} [\rho \delta_{S_i,1} + (1-\rho)\delta_{S_i,-1}] \\ &= \frac{1}{(e^{-h}+e^h)^N} e^{h \sum_{i=1}^N S_i}, \end{aligned} \tag{8.2}$$

where the first part of the second equation is in fact S-independent and find its way into the partition function Z, to finally find (8.3).

[16] For completeness note that the bracket in the exponent of equation (8.1) is multiplied out and then simplified by taking advantage of the fact that $S_i \in \{-1,1\}$, $Y_{ij} \in \{-1,1\} \forall i,j$ and that the squares of these are always equal to one.

- If $\hat{p} > 1$, then accept $\vec{S'}$
- If $\hat{p} < 1$, then accept $\vec{S'}$ with probability $p = \hat{p}$.

8.1.2 Gibbs Sampling/Heat Bath

The *Gibbs sampling* algorithm, or *heat bath* in statistical physics, is another common example of a Monte Carlo Markov Chain algorithm. Given the configuration $\{S_i\}$, we pick one variable S_k, and sample S_k from the conditional probability $P(S_k|\{S_i\}_{i\neq k})$, while keeping the values of the other variables fixed. For the spin glass card game – in which there are only two values for S_k, ± 1 – the conditional probability is as follows:

$$P(S_k|\{S_i\}_{i\neq k}) = \frac{e^{\frac{1}{\Delta}H_k S_k}}{e^{\frac{1}{\Delta}H_k} + e^{-\frac{1}{\Delta}H_k}} \tag{8.6}$$

where Δ is the noise variance[17] and $H_k = \sum_j S_j Y_{jk} \frac{1}{\sqrt{N}}$.

Thermodynamically, the "heat bath" can be accounted for H_k playing the role of the local magnetic field. All the spins but S_k represent the bath in which the latter is. In contrast with the Metropolis-Hasting rule, Gibbs sampling has no rejections and can be also easily used for continuous variables. Nowadays, Gibbs sampling is quite popular in Machine Learning. Both the Gibbs sampling and the Metropolis-Hastings algorithm can be used to compute the local magnetisation and the MMSE. When coming to the convergence time there exist no usable general rules. It may be be that the algorithms converge in $\exp(n)$ only or very fast. In general, however, it is worth trying it out.

8.1.3 Simulated Annealing

We recall that we have seen in the last lecture that the **maximum a posteriori estimator** (MAP) is equivalent to finding the ground state of the Hamiltonian associated to the system. How can we use the MCMC to find this ground state? For a general system, the probability of a given state at constant temperature T is given by the Boltzmann distribution:

$$P(S) = \frac{1}{Z} e^{\frac{-\mathscr{H}(S)}{T}}.$$

When the temperature drops to zero, the measure will concentrate on the minimum of the Hamiltonian. Indeed, lowering the temperature reduces thermal fluctuations. Then, one may think of initializing a system at low temperature and run the MCMC until equilibrium to sample the ground state. Unfortunately, this is inefficient, since having low temperatures reduces the rate at which the system evolves. Therefore, another idea would be to initialize the system at high temperature and then to drop the temperature until it is small. This is a good idea, but with some caveats. When dropping the temperature too quickly, the system may not settle into the ground state at all and may remain trapped in a high-temperature state (a local minimum of the Hamiltonian). This fast temperature decrease is called "quenching". To avoid this problem, the temperature must be lowered slowly so that the system can reach the ground state – a process known as *annealing*.

Let us see formally how this is done. First, let us initialize the system at a large temperature $T^0 \gg 1$. We then compute a MCMC step for selecting the next state. After the step, we slightly lower the temperature by a multiplicative factor $0 < \delta < 1$, i.e.,

$$T^1 = \delta T^0,\ \delta \in (0,1).$$

Then, we do another Monte Carlo step and lower the temperature again. At step i,

$$T^{i+1} = \delta T^i = \delta^i T^0.$$

Provided the temperature is lowered slowly enough, the system should converge to the ground state of the Hamiltonian. This process has a physical analogue.

To see an example of quenching in a real system, we look towards the process of hardening metals. In this process, a metal is heated until it is red-hot before being dipped in cold oil or water. The rapid temperature change causes the outside of the metal to contract much faster than the inside, and tension is introduced into the metal. This is an energized state, and it is most certainly not the same state the metal was in before it was heated. To remove or weaken the hardening on a metal, one can reheat it (often to a temperature much lower than the one used for the quench) and then annealed it – slowly cooling it down at room temperature. This prevents the system from being trapped into an excited state, since it relaxes globally.

[17] Note that by making the link to statistical physics one would need to set $\Delta = \frac{1}{\beta}$.

8.2 Bayesian Parameter Estimation

We have already discussed the biased card game, where the probability of $S_i = +1$ is ρ, i.e., $P(S_i = +1) = \rho$, and the probability of $S_i = -1$ is $1 - \rho$, i.e. $P(S_i = -1) = 1 - \rho$. We do not necessarily have that $\rho = 0.5$. Now, what if ρ is not known? The posterior probability reads:

$$P(\rho|Y) = \frac{P(Y|\rho)P(\rho)}{P(Y)} = \frac{P(\rho)}{P(Y)} \sum_{\{S_i\}_{i=1}^N} P(Y, \{S_i\}|\rho) = \frac{P(\rho)}{P(Y)} \sum_{\{S_i\}_{i=1}^N} P(Y|\{S_i\}, \rho)P(\{S_i\}|\rho), \tag{8.7}$$

where we have used Bayes theorem (Eq.(2.11)) and the law of total probability. If we assume that we know the model $Y_{ij} = \frac{S_i^* S_j^*}{\sqrt{N}} + W_{ij}$, where W_{ij} is a symmetric matrix with noise sampled from a normal distribution with variance Δ, we can write:

$$P(\rho|Y) = \frac{P(\rho)}{P(Y)} \sum_{\{S_i\}_{i=1}^N} \prod_{i<j} \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}(Y_{ij} - \frac{S_i S_j}{\sqrt{N}})^2} \prod_{i=1}^N [\rho\delta(S_i - 1) + (1 - \rho)\delta(S_i + 1)] \tag{8.8}$$

The term $\sum_{\{S_i\}_{i=1}^N} \prod_{i<j} \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}(Y_{ij} - \frac{S_i S_j}{\sqrt{N}})^2} \prod_{i=1}^N [\rho\delta(S_i - 1) + (1 - \rho)\delta(S_i + 1)]$ is called the normalization of the posterior $P(S|Y)$, that we denote $Z(\rho)$, which is also known as the evidence of ρ. Now, $Z(\rho)$ will be exponential in N, but $P(\rho)$ will be of order 1. Thus, we can disregard $P(\rho)$ for large N (for $N \to \infty$, the ML estimator and the MAP estimator are the same).

Making a comparison with physics, we can define the "free energy" $f(\rho) = -\frac{1}{N}\log(Z(\rho))$. The maximum likelihood estimator will be the maximum of the evidence over ρ, or, equivalently, the minimum of the free energy $f(\rho)$. In particular, this will be a good estimator if N is large. This method can be used in general to learn hyperparameters. To find an estimation of a hyperparameter θ, compute the normalization $Z(\theta)$, and maximize $Z(\theta)$ over θ.

8.2.1 Expectation Maximization (for learning parameters)

LZ: A better example for this section is the one done in class in 2022 where $P(S_i) = \rho[\delta(S_j - 1) + \delta(S_j + 1)]/2 + (1 - \rho)\delta(S_j)$

So we would like to maximize $Z(\rho)$ efficiently. Let us start with finding the maximum by imposing the derivative of evidence Z being equal to zero:

$$\begin{aligned} 0 &= \frac{\partial}{\partial\rho}\log Z(\rho) = \\ &= \frac{1}{Z(\rho)} \sum_{\{S_i\}_{i=1}^N} \sum_{j=1}^N \frac{\delta(S_j - 1) - \delta(S_j + 1)}{\rho\delta(S_j - 1) + (1 - \rho)\delta(S_j + 1)} \underbrace{\prod_{i=1}^N [\rho\delta(S_i - 1) + (1 - \rho)\delta(S_i + 1)] \prod_{i<j} e^{-\frac{1}{2\Delta}\left(Y_{ij} - \frac{S_i S_j}{\sqrt{N}}\right)^2}}_{\text{Boltzmann measure } P(S|Y)}. \end{aligned} \tag{8.9}$$

Recall the definition of the marginal distribution (cf. Eq.(7.17)):

$$\mu(S_j) = \sum_{\{S_i\}_{i\neq j}} P(S|Y). \tag{8.10}$$

By substitution of (8.10) into (8.9) and using the fact that $\sum_{\{S_i\}_{i=1}^N} = \sum_{S_j} \sum_{\{S_i\}_{i\neq j}}$ one can obtain:

$$0 = \sum_{j=1}^N \sum_{S_j} \mu(S_j) \frac{\delta(S_j - 1) - \delta(S_j + 1)}{\rho\delta(S_j - 1) + (1 - \rho)\delta(S_j + 1)} = \sum_{j=1}^N \left[\mu(S_j = +1)\frac{1}{\rho} + \mu(S_j = -1)\frac{-1}{1 - \rho}\right]. \tag{8.11}$$

Let's rewrite the expression in terms of m_i, the local magnetization:

$$m_i = \sum_{S_i} S_i \mu(S_i) = \mu(S_i = +1) - \mu(S_i = -1). \tag{8.12}$$

Also, due to the definition of $\mu(S_i)$:

$$\mu(S_i = +1) + \mu(S_i = -1) = 1. \tag{8.13}$$

Combining these two equations together we obtain:

$$\mu(S_i = \pm 1) = \frac{1 \pm m_i}{2}. \tag{8.14}$$

It is convenient to introduce also the total magnetization $m = \frac{1}{N}\sum_{i=1}^{N} m_i$. Then:

$$\frac{1}{N}\sum_{i=1}^{N} \mu(S_i = \pm 1) = \frac{1 \pm m}{2}. \tag{8.15}$$

Finally, expression (8.11) becomes:

$$0 = \frac{1}{\rho}\frac{1}{N}\sum_{i=1}^{N} \mu(S_i = +1) - \frac{1}{1-\rho}\frac{1}{N}\sum_{i=1}^{N} \mu(S_i = -1) = \frac{1+m}{2}\frac{1}{\rho} - \frac{1-m}{2}\frac{1}{1-\rho}. \tag{8.16}$$

The solution of this equation is:

$$\rho = \frac{1+m}{2} = \frac{1}{N}\sum_{i=1}^{N} \mu(S_i = +1). \tag{8.17}$$

So ρ can be estimated just as the average of all marginals with $S_i = +1$. However, we actually need ρ to calculate marginals $\mu(S_i)$. We could solve this issue by using the following iterative algorithm called expectation maximization:

1. start with some initial value $\rho^{t=0} = \rho_0$;
2. make step $\rho^{t+1} = \frac{1+m(\rho_t)}{2}$;
3. check convergence and repeat if necessary.

To summarize, we found the generic procedure to learn a set of parameters by iteratively imposing expectation consistency conditions such as $\rho = \frac{1+m}{2}$.

9 Lecture 9: Clustering & Boltzmann Machine 19.11.

Students designated to write this section: He Shengyu, Haji Mirsaeedi Sayed Mohammad, Griffon Amaury Brice Marie, Golomer Bastien Olivier Yves Francis, Gollerthan Tim
Coordinator:
TA: Federica (TA approved)
Students who actually contributed: He Shengyu, Haji Mirsaeedi Sayed Mohammad, Golomer Bastien Olivier Yves Francis, Gollerthan Tim, Griffon Amaury Brice Marie

9.1 Clustering

When analyzing a set of objects, it is often helpful to distinguish them in different categories, by putting similar objects into the same group. This is called *clustering*. By this method, we can gain a deeper understanding of the data structure. A big advantage of this process is its predictive power. Since all objects in one group share some general properties, we can infer these properties by grouping new data. One could for example distinguish all animals in fish, birds and land animals. If a new animal is similar to a fish, we can, for example, immediately predict that it can probably swim.

9.1.1 General Treatment

In order to treat these concepts in a more general way, we will now consider a set $X \in \mathbb{R}^{n\times d}$ of unlabelled data, i.e., a set of $n \in \mathbb{N}$ objects with $d \in \mathbb{N}$ features respectively. In terms of unsupervised learning, we are interested in discovering structures in X by dividing it in $k \in \mathbb{N}$ disjoint clusters S_a with $a = 1, \dots, k$. The set $S = \{S_a\}_{a=1}^{k}$ is called a clustering of X. The number of clusters k is normally treated as another hyperparameter to fine-tune. However, for the sake of simplicity, in the following we will assume k to be given. An example of data-points structured in clusters is given in Figure 21 for the case $d = 2$ and $k = 3$.

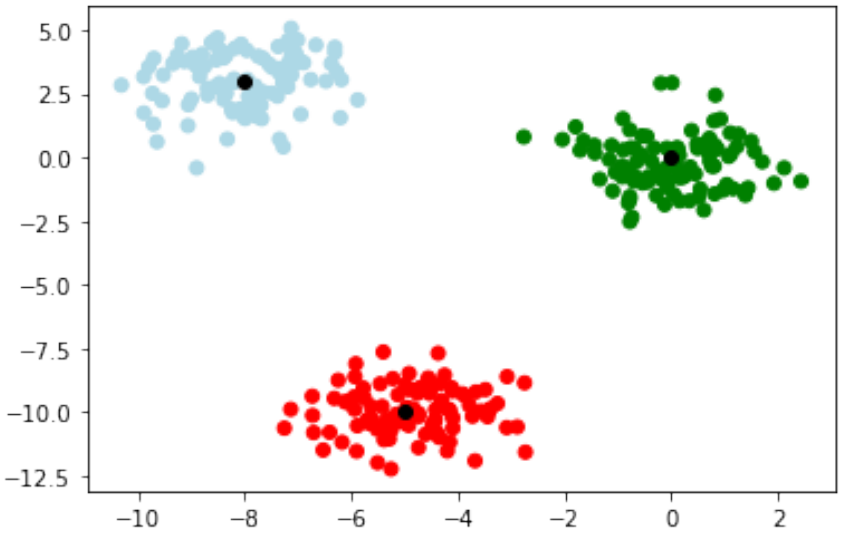


Figure 21: Example of clustering in case $d = 2$ and $k = 3$. The different colours refer to different cluster membership. The black dots correspond to each cluster centre.

In order to identify a clustering-type of structure in the input data, we need a rule to decide which data point belongs to which cluster. This is given by defining a loss function $\mathscr{L}(S)$, providing a sort of similarity measure between one data-point and all the other data points belonging to a given cluster. The optimal clustering S will then correspond to the configuration minimizing $\mathscr{L}(S)$. A meaningful choice for $\mathscr{L}(S)$ is for example:

$$\mathscr{L}(S) := \frac{1}{n} \sum_{a=1}^{k} \sum_{X_\mu \in S_a} \|X_\mu - u_a\|^2, \tag{9.1}$$

where the similarity measure is defined through the Euclidean distance between one data point X_μ and the centre u_a of each cluster S_a, defined as:

$$u_a := \frac{1}{|S_a|} \sum_{X_\mu \in S_a} X_\mu. \tag{9.2}$$

Equation (9.1) can be seen as follows. For each data point X_μ, we measure its distance to each cluster centre u_a and then make a sum over all clusters. The cluster assignment $S^\star$ minimizing the loss function is the one where each data point X_μ belongs to the cluster with the closest centre. Note, that the clustering highly depends on the distance measure and the choice of cluster centres. In general, a variety of distance measures is possible, and one should select a measure which best suits for a specific problem.

9.1.2 k-Means Algorithm

In order to minimize the cost function defined in Eq.(9.1) and therefore to identify a meaningful clustering, we need an algorithm which can gradually detect each cluster centre. This algorithm goes under the name of k-Means and is defined in the following pseudo-code:

1. Initialize k centres $u_1^{(0)}, \dots u_k^{(0)} \in \mathbb{R}^d$.
   (E.g. choose $u_a^{(0)} = X_\mu$ for a random X_μ and each $a = 1, \dots k$).
2. Assign each sample X_μ to the closest centre.
3. Update the centres to $u_a^{(t+1)} = \frac{1}{|S_a^{(t)}|} \sum_{X_\mu \in S_a(t)} X_\mu$.
   (In case $S_a = \emptyset$ keep $u_a^{(t+1)} = u_a^{(t)}$).
4. Repeat step 2 and 3 until convergence or stopping time is reached.

Convergence is in general not guaranteed and in case the algorithm converges the result is not necessarily a clustering that minimizes $\mathscr{L}(S)$. The reason for this is that minimizing the loss $\mathscr{L}(S)$ is computationally hard. However, the k-Means algorithm is a simple heuristic one that often works well. Another thing to notice is that the algorithm depends on the choice of cluster centres for initialization. The simplest choice is to initialize the centres by selecting random points as initial cluster centres from the training set. This procedure is satisfying in many cases, but it is still not the best thing to do. There are indeed other initialization schemes that ensure better performances. Of course, if we make a particular choice, for instance selecting the first cluster centre as one of the data points and the other centres very far away from the actual data, the algorithm will not find a good clustering. For instance, in this case, it would give $S_1 = X$ and $S_2 = \emptyset, \dots S_k = \emptyset$ as clustering. Therefore, one can benefit from a better choice of cluster centres, i.e., some intuition about the data.

9.1.3 Probabilistic Model: Gaussian Mixture Model

At this point, one can ask a more fundamental question: Why are we using the specific loss function in Eq.(9.1)? To answer this question, we should go back to the one-hot-encoding introduced in Lecture 5, when dealing with multi-class classification problems. Indeed, one-hot encoding can also be used to specify whether a point X_μ belongs to cluster S_a or not. In particular we select vectors $v_\mu \in \mathbb{R}^k$ such that $v_{\mu a} = 1$ if $X_\mu \in S_a$ and $v_{\mu a} = 0$ otherwise for all $a = 1, .., k$. Hence the v_μ belong to the k-dimensional canonical basis $\{e_a\}_{a=1}^k$, used for one-hot-encoding. With this new basis we can rewrite the loss function $\mathscr{L}(S)$ as:

$$\mathscr{L}(S) := \frac{1}{n}\sum_{a=1}^{k}\sum_{X_\mu \in S_a} \|X_\mu - u_a\|^2 = \frac{1}{n}\sum_{\mu=1}^{n}\sum_{i=1}^{d}\left(X_{\mu i} - \sum_{a=1}^{k} u_{ai} v_{\mu a}\right)^2 \tag{9.3}$$

This notation puts in better evidence the underlying model of structured data in clustering problems, going under the name of *Gaussian Mixture model*. In this model, v^* are the true cluster memberships, taken uniformly from $\{e_a\}_{a=1}^k$, and u^* are the true cluster centres, taken for instance from normal distributions. According to this model, we generate synthetic data as:

$$X_{\mu i} = \frac{1}{\sqrt{d}}\sum_{a=1}^{k} v^*_{\mu a} u^*_{ai} + \xi_\mu \tag{9.4}$$

where v^*_μ is the ground-truth one-hot vector, encoding the cluster memberships of data-point X_μ and taken uniformly from $\{e_a\}_{a=1}^k$, while u^*_a is the ground-truth centre of cluster a, sampled for instance from the normal distribution. The additional term $\xi_\mu \sim \mathscr{N}(0, \sigma)$ corresponds to the noise, here assumed to be the same for all clusters.

According to this model, we first take a centre, we then model each of the samples $X_{\mu i}$ that belong to the cluster corresponding to this centre as this centre plus Gaussian noise. We can then pick k of these centres and built k different Gaussian "clouds". For large d, the $\frac{1}{\sqrt{d}}$ corresponds to the scaling into which the model is interestingly behaving.

If the variance σ of the noise is small, it is easy to see which point belongs to which cluster. If it is large, then all the clusters will start to merge. As in the spin glass card game, we may have a phase transition in the value of the σ where the clusters become indistinguishable.

To estimate the ground truth v^*, u^* from the knowledge of X, we compute the posterior as exemplified in Lecture 3:[18]

$$P(u, v \mid X) = \frac{1}{Z}P(u)P(v)\prod_{\mu=1}^{n}\prod_{i=1}^{d} e^{-\frac{1}{2\sigma}\left(X_{\mu i} - \frac{1}{\sqrt{d}}\sum_{a=1}^{k} u_{ia} v_{\mu a}\right)^2}. \tag{9.7}$$

By transforming the product of exponentials into a single exponential whose exponent is the sum of the exponents of the single exponentials, we can then recognize the loss function $\mathscr{L}(S)$ eq. (9.3) at the exponent. This means that, as we have shown for linear regression in chapter 3, even for clustering the choice of the cost function $\mathscr{L}(S)$ can be

[18] Mathematically we have in this case the likelihood:

$$P(X|u, v) = \frac{1}{Z}\prod_{\mu=1}^{n}\prod_{i=1}^{d} e^{-\frac{1}{2\sigma}(X_{\mu i} - \frac{1}{N}\sum_{a=1}^{k} u^*_{ia} v^*_{\mu a})^2} \tag{9.5}$$

and the prior:

$$P(v). \tag{9.6}$$

The evidence can be seen in (9.7) as being $\frac{1}{Z}$.

justified in terms of Bayesian inference through a probabilistic model of synthetic data, namely the Gaussian Mixture one. Optimizing the cost function $\mathscr{L}(S)$ then corresponds to optimize the log-likelihood once again.

Remarks

- In Lecture 7, we talked about Bayesian estimation. We said that if we have a good model, it would give better results than the maximum likelihood estimator or than the MAP estimator. This works as well with the Gaussian Mixture model. Sampling the posterior would give a better overlap with the ground truth.
- We note that the Gaussian Mixture model together with the spin glass card game are both examples of low-rank matrix estimation problems. The number of clusters k would be the rank of these matrices.
- According to the way we wrote the probabilistic model so far, the noise ξ_μ are Gaussian random variables with identical variance σ. This would mean that the data-points X_μ are sampled according to the Gaussian probability distribution:
$$P(X_\mu) = \prod_{a=1}^{k} \prod_{\mu \in S_a} \frac{e^{-\frac{1}{2}(X_\mu - m_{\mu a})^t \Sigma^{-1} (X_\mu - m_{\mu a})}}{(2\pi)^{d/2} \left(\det \Sigma\right)^{1/2}}, \tag{9.8}$$
with $m_{\mu a} = \frac{1}{\sqrt{d}} \sum_{a=1}^{k} v_{\mu a} u_a$, $\Sigma = \sigma \mathbb{I} \in \mathbb{R}^{d \times d}$ and $\mathbb{I}$ being the identity matrix. This corresponds to the case where the shape of the clusters looks homogeneous and isotropic (i.e., hyperspheres). However, in principle we could consider generic covariance matrices Σ, which would take into account non-homogeneous and anisotropic clusters (i.e., hyperellipsoids);
- The size of the clusters is encoded in the prior: $P(v) = \prod_{i=1}^{d} \left(\sum_{a=1}^{k} \rho_a \delta(e_a - v) \right)$ where $\sum_a \rho_a = 1$ (ρ_a are the fractions corresponding to the size of cluster a).

9.2 Generative Models

Unsupervised learning can be seen in two main ways (i.e., can have two related goals):

1. Uncover structure in unlabelled data. This will allow dimensionality reduction, clustering (recall PCA and the spin glass card game).
2. Learn a way to generate the data that look like the ones we observe: These are called generative models.

To recapitulate the difference between supervised and unsupervised learning, let us have a look at a dataset of pictures of cats and dogs. Let's call X_{train} the training set, X_{test} the testing set.

- Supervised: In addition to X_{train}, the training data contain the correct labels y_{train}. This allows to train the model to predict the right label for a new data sample in the test set.
- Unsupervised: the model has only access to X_{train} and no labels. As explained above, the model can be asked to either recognise patterns or to generate new data samples (new pictures of cats or dogs for instance).

In the next section, we will deal with unsupervised learning in the context of generative models. In particular we will focus on a generative model which goes under the name of Boltzmann Machine.

9.2.1 Boltzmann Machine

To understand what is *Boltzmann Machine*, let us first define its goal. We have data $X \in \mathbb{R}^{n \times d}$, all the elements are considered binary $X_{\mu i} \in \{\pm 1\}$, we have n samples of d-dimensional sequences of strings of plus or minus one, i.e., $X_\mu \in \{\pm 1\}^d$. The goal of the Boltzmann Machine is to generate a new string that looks like one of the strings of the data. For example: If X_μ is often characterized by a $+1$ in its third position, the new string X_{new} should also have a $+1$ in its third position. To sum up, if a set of strings is characterized by some properties, then the new string should be characterized by those properties too.

Now let's see how the Boltzmann machine works. We first consider a probability distribution:

$$p_{h,J}(x) = \frac{1}{Z} e^{\sum\limits_{i<j} J_{ij} x_i x_j + \sum\limits_{i=1}^{d} h_i x_i} \tag{9.9}$$

with x being the d-dimensional vector such that $x \in \{\pm 1\}^d$. To distinguish, we denote by capital $X_{\mu i}$ the observed data and by small x the variable of the generative model.

The Boltzmann Machine aims to generate new data points, distributed according to the Boltzmann measure above. The goal is then to infer the couplings J_{ij} and the field h_i from a set of samples. Once the J_{ij} and h_i have been inferred, we can then use the Boltzmann measure to sample (generate) new data points. This method is also called the inverse Ising model. Indeed, in the Ising model, we have the interactions and the magnetic field, and we then try to describe the probability distribution, while here we start with samples from the probability distribution and we then try to infer the interactions and magnetic field. That is why it is the inverse.

Having defined the goal of Boltzmann Machines, we may have three questions we would like to address: First, how do we infer both the couplings and the magnetic field? Second, why are we considering the Boltzmann measure and not some other probability distribution? Third, once we have inferred the couplings and the magnetic field, how do we sample from the Boltzmann measure?

The last question is easy to answer: We can run Monte Carlo as seen in the previous sections. Instead, to answer the first question, the strategy to estimate J_{ij} and h_i will be to first express the two quantities:

$$\begin{aligned} \mu_i &= \frac{1}{n}\sum_{\mu=1}^{n} X_{\mu i} \\ \Sigma_{ij} &= \frac{1}{n}\sum_{\mu=1}^{n} X_{\mu i} X_{\mu j} \end{aligned} \tag{9.10}$$

where μ_i is the empirical mean counting how often the strings have a $+1$ or -1 on the i^{th} position, meanwhile Σ_{ij} is the empirical correlation between the i^{th} and j^{th} position, counting how often position i and position j are the same. These two quantities are directly computed from the observed data. However, the same quantity can be derived from the Boltzmann measure Eq.(9.9), by assuming that the data have been sampled from this specific probability distribution: $\langle x_i \rangle_{p_{h,J}(x)}$ and $\langle x_i x_j \rangle_{p_{h,J}(x)}$. These values depend on J_{ij} and h_i because the value of the magnetization depends on the interactions and the magnetic field. The idea is to select the h_i and J_{ij} so that the empirical mean μ_i and correlation Σ_{ij} corresponds to the mean and correlation computed under the Boltzmann measure:

$$\begin{aligned} \langle x_i \rangle_{p_{h,J}(x)} &= \mu_i \,, \\ \langle x_i x_j \rangle_{p_{h,J}(x)} &= \Sigma_{ij} \,. \end{aligned} \tag{9.11}$$

We can ask ourselves why is this the right thing to do, why do we choose this type of probability distribution? We will answer this question in the next section by means of the maximum entropy principle.

9.2.2 Maximum Entropy Principle

To justify the choice of a Boltzmann-like probability distribution, we start from the definition of the Shannon entropy. The Shannon entropy quantifies the average level of uncertainty associated to the possible outcomes of a given random variable x and it is defined in the light of the probability distribution $p(x)$ as:

$$S = -\sum_x p(x) \log p(x) \,, \tag{9.12}$$

where the sum runs over the number of all possible configurations of x (i.e., over 2^d of them in the case of binary variables). Given this definition, assume we have a set of observed quantities $f_a(x)$, with $a = 1, \ldots, m$, $m = d + \frac{d-1}{2} d$, that we want to fix (the index a here runs over the number of constraints we want to impose). These observables may be some quantities that we want to conserve. For instance, we would like to impose the constraint that the empirical means are as close as possible to the expectations over a desired probability distribution $p(x)$:

$$\langle f_a \rangle_{\text{emp}} = \sum_x f_a(x) p(x) = \langle f_a \rangle_{p(x)} \,. \tag{9.13}$$

Given this set of constraints, we then would like to choose $p(x)$ in such a way that all the constraints in Eq.(9.13) are actually satisfied. In principle, there are many different probability distributions that can give us the right mean of $f_a(x)$ $\forall a$. The **maximum entropy principle** (MEP) tells us how to chose the most reasonable one among them. The idea of MEP is to pick the probability distribution that maximizes the Shannon entropy under the constraints in Eq.(9.13). To solve this problem, we use the method of Lagrange multipliers. According to this method, we first

need to define the Lagrangian functional as the sum of the quantity that we aim to maximize (the Shannon Entropy in this specific case) and the set of constraints we aim to satisfy:

$$\mathscr{L}(p,\lambda,\gamma) = -\sum_x p(x)\log p(x) + \sum_{a=1}^{m}\lambda_a\Big(\langle f_a\rangle_{\text{emp}} - \sum_x f_a(x)p(x)\Big) + \gamma\big(1-\sum_x p(x)\big)\,. \tag{9.14}$$

In particular, the last constraint involving the Lagrangian parameter γ ensures that the probability distribution $p(x)$ is normalized to one, as all probability distributions should be. Having defined the Lagrangian, we can determine the probability distribution maximizing the Shannon entropy under the desired constraints by differentiation of the Lagrangian functional:

$$\frac{\delta\mathscr{L}}{\delta p(y)} = 0 = -\log p(y) - 1 - \sum_{a=1}^{m}\lambda_a f_a(y) - \gamma \tag{9.15}$$

This will give rise to a Boltzmann-like type of measure:

$$p(x) = e^{-\gamma-1}e^{-\sum\limits_{a=1}^{m}\lambda_a f_a(x)} = \frac{1}{Z}e^{-\sum\limits_{a=1}^{m}\lambda_a f_a(x)}\,. \tag{9.16}$$

Note that, for $m = 1$ (e.g. just one constraint) and in the case where the only quantity we are constraining is the energy function $E(X)$, the resulting probability distribution is:

$$p(X) = \frac{1}{Z}e^{-\lambda E(X)}\,. \tag{9.17}$$

If we rename λ by calling it β, then $p(X)$ is nothing but the Boltzmann distribution:

$$p(X) = \frac{1}{Z}e^{-\beta E(X)}\,. \tag{9.18}$$

This derivation is of interest also because it shows us where the exponential form of the Boltzmann measure comes from: It arises from the maximization of the Shannon entropy while keeping the energy fixed.

In the specific case of the Boltzmann Machine, we have more than one constraint. Those constraints are the ones in Eq.(9.11), where:

$$\begin{aligned} &f_a(x) = x_i, \qquad a = i = 1,\dots,d\,,\\ &f_a(x) = x_i x_j, \qquad a = ij = d+1,\dots,d(d+1)/2\,, \end{aligned} \tag{9.19}$$

with the Lagrange multiplier being $\lambda_a = -h_a$ for $a = 1,\dots,d$ and $\lambda_a = -J_{ij}$ for $a = d+1,\dots,d(d+1)/2$. This is why we are working with Boltzmann measures.

One further question which may arise is why we want to precisely maximize the entropy and not some other quantity. In physics we know that if we would sample uniformly from the space of all possible configuration of x, the probability of getting those configurations which are actually maximizing the entropy is much higher than the ones of other configurations.

9.2.3 How to train the Boltzmann Machine

After the introduction of Boltzmann Machine, the next question we think about is how to train the Boltzmann Machine and how to find the values of h_i, J_{ij}. We basically have different ways to accomplish this task.

The first strategy is that we simply relax the requirement that $x \in \{\pm 1\}^{n\times d}$ and consider x to belong to real numbers. Then the $p(x)$ can be expressed by a multivariate Gaussian function as:

$$p(x) = \frac{1}{Z}e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)} \tag{9.20}$$

where Z is the normalization factor, namely $Z = (2\pi)^{\frac{d}{2}}(\det\Sigma)^{\frac{1}{2}}$, while the mean μ and the covariance Σ of the multivariate Gaussian distribution are:

- $\mu_i = \langle x_i\rangle_{p(x)} = \frac{1}{n}\sum\limits_{\mu=1}^{n} X_{\mu i},\ \ \mu\in\mathbb{R}^d$
- $\Sigma_{ij} = \langle x_i x_j\rangle_{p(x)} = \frac{1}{n}\sum\limits_{\mu=1}^{n} X_{\mu i}X_{\mu j}\ ,\ \ \Sigma\in\mathbb{R}^{d\times d}$

By comparing this expression of $p(x)$ (9.20) with the one provided in Eq.(9.9), we conclude:

$$h = -\Sigma^{-1}\mu,\ J = -\Sigma^{-1}\,. \tag{9.21}$$

Although this method seems simple and naive, we can still obtain quite a good approximation for both the couplings J and the field h. This method is often used in the direct coupling analysis of protein sequences (see e.g. the work of De Los Rios and Goloubinoff at EPFL [15]).

Another training strategy is more common in Machine Learning and once again involves the maximum likelihood principle. In our case, the likelihood is nothing but the product over all samples of the Boltzmann-measure defined in Eq.(9.9). Indeed, the Boltzmann measure is quantifying the probability of having observed a certain configuration, given the couplings J and the magnetic field h:

$$\mathscr{L}(X|\theta) = \prod_{\mu=1}^{n} p_{h,J}(X_\mu) = \frac{1}{Z(\theta)^n} e^{\sum_{\mu=1}^{n} E_\theta(X_\mu)}\,. \tag{9.22}$$

In this expression, we use the probability $p_\theta(x) = \frac{1}{Z(\theta)} e^{E_\theta(x)}$ where the θ represents the set of (h, J) parameters we aim to infer, $x \in \mathbb{R}^d$, and the energy is given by $E_\theta(x) = \sum_{i<j} J_{ij}x_i x_j + \sum_i h_i x_i$. We can then maximize the likelihood over θ by taking the logarithm:

$$\max_\theta \left(\log \mathscr{L}(X|\theta)\right) = \max_\theta \left(\frac{1}{n}\sum_{\mu=1}^{n} E_\theta(X_\mu) - \log Z(\theta)\right). \tag{9.23}$$

At the maximum we have:

$$\frac{\partial \log(\mathscr{L}(X|\theta))}{\partial\theta} = 0 = \left\langle \frac{\partial E_\theta(X)}{\partial\theta}\right\rangle_{\text{emp}} - \frac{1}{Z(\theta)}\sum_x e^{E_\theta(x)}\frac{\partial E_\theta(x)}{\partial\theta} = \left\langle \frac{\partial E_\theta(X)}{\partial\theta}\right\rangle_{\text{emp}} - \langle \frac{\partial E_\theta(x)}{\partial\theta}\rangle_{p_{h,J}(x)}. \tag{9.24}$$

Given the dependency of the energy function E_θ on the couplings J_{ij} and the magnetic field h_i, $\forall J_{ij}, h_i$, Eq.(9.24) can be re-written as:

- $\langle X_i\rangle_{\text{emp}} = \langle x_i\rangle_{p_{h,J}(x)}$,
- $\langle X_i X_j\rangle_{\text{emp}} = \langle x_i x_j\rangle_{p_{h,J}(x)}$,

which are nothing but the constraints we aim to satisfy. We can achieve this condition algorithmically, by maximizing the log-likelihood $\log(\mathscr{L}(X|\theta))$ through gradient descent:

$$\theta^{t+1} = \theta^t - \gamma \frac{\partial(-\log(\mathscr{L}(X|\theta)))}{\partial\theta}\Big|_{\theta^t} \tag{9.25}$$

where γ represents the learning rate. If we re-write the gradient descent update in terms of h and J, we then get the following update rules:

- $h_i^{t+1} = h_i^t + \gamma[\langle X_i\rangle_{\text{emp}} - \langle x_i\rangle_{p_{h^t,J^t}(x)}]$,
- $J_{ij}^{t+1} = J_{ij}^t + \gamma[\langle X_i X_j\rangle_{\text{emp}} - \langle x_i x_j\rangle_{p_{h^t,J^t}(x)}]$.

Now that we have an algorithm to infer both the couplings and the field, the last thing to understand is how to compute the empirical means and the expectations appearing in the update rules. As for the empirical means, it is easier because we can estimate them directly from the data. However, to compute the expectations $\langle x\rangle_{p(x)}$ over the model at fixed J and h, we need to run Monte Carlo Markov Chain (MCMC). For the details on MCMC refer Lecture 7 and 8.

To conclude, this way to train the Boltzmann Machine is based on the maximum likelihood principle. To infer the parameters, we use the gradient descent algorithm. Even though gradient descent provides us the update rules to determine the parameters, to get the final result we also need to run MCMC to estimate the expectations over the Boltzmann measure. So, we see the process of training Boltzmann Machine is really combining several methods we learnt before (both GD & MCMC) to be able to infer the parameters of the Boltzmann distribution, which we can then use to produce new samples distributed according to the same Boltzmann measure.

10 Lecture 10: Features, Kernel methods 26.11.

Students designated to write this section: Girod Alexis Théo Allan, Giriens Matthieu Philippe, Galletti Chiara, Forster Benjamin John Morcombe, Ferrari Filippo, Farchy Tobias Thagi Mckee
Coordinator: Filippo Ferrari
TA: Giovanni (TA approved)
Students who actually contributed: Girod Alexis Théo Allan, Giriens Matthieu Philippe, Galletti Chiara, Forster Benjamin John Morcombe, Ferrari Filippo, Farchy Tobias Thagi Mckee

10.1 Introduction on non-linearly separable data

So far, linear regression and classification have been discussed. These were useful when the data is linearly separable. However, sometimes it is not, and one needs to think about different methods of classification. The k-nearest neighbours is one of these, it can "draw" non-linear separations, but it suffers from the dimensionality curse discussed previously. So one may ask the question: **What else to do ?**

Take the case of data as in Figure 23, what would be a good idea for separating these data? One solution would be to draw a circle that would separate the two classes in the data. This can be achieved by introducing a new *feature* $\varphi(x_1, x_2)$:

$$\varphi(x_1, x_2) = (x_1^2, x_2^2)\,. \tag{10.1}$$

In the new space, data look like in Figure 25, and so are now linearly separable. A linear or a logistic regression would classify them very efficiently.

Another non-linearly separable dataset is the XOR-like function, represented in Figure 22. Which transformation makes these data linearly separable? One sees easily that with the following feature:

$$\varphi(x_1, x_2) = (x_1 x_2). \tag{10.2}$$

The black circles get a negative coordinate while the blue squares a positive one. So, the point $\varphi(x_1, x_2) = 0$ separates efficiently the data as in Figure 24.

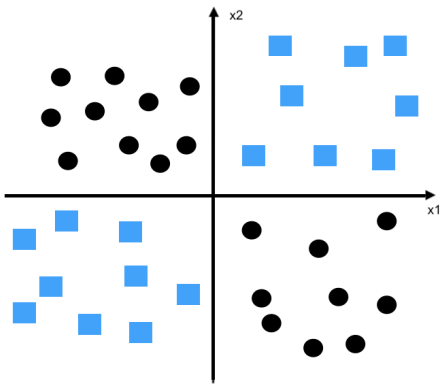


Figure 22: Example of non-linearly separable data.

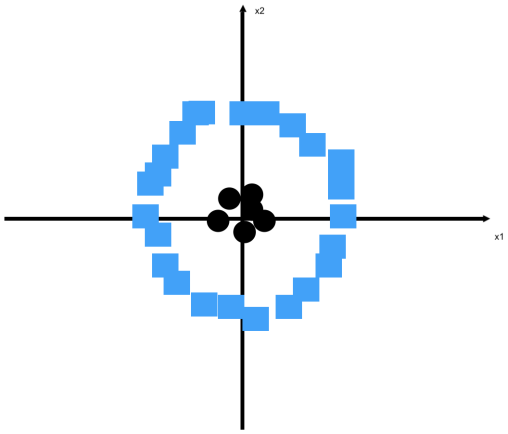


Figure 23: Example of non-linearly separable data.

We see that the general idea of non-linear regression methods is to make data linearly separable by adding *features*. If $X \in \mathbb{R}^d$ is one data point, we transform this point by using a map $\varphi : \mathbb{R}^d \to \mathbb{R}^p$ and then we do regression on $\{\varphi(X_\mu), y_\mu\}_{\mu=1}^n$. We have already met a regression with features: The polynomial regression, in that case we have defined $(d = 2)$:

$$\varphi(X) = (x_1, x_2, x_1^2, x_1 x_2, x_2^2, ...)\,. \tag{10.3}$$

Figure 24: Representation of data from Fig. 22 in the direction of the new feature $\varphi(x_1, x_2) = (x_1 x_2)$.

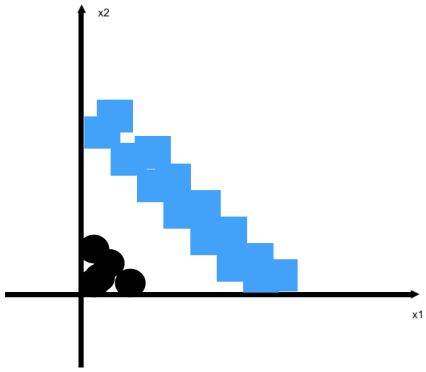


Figure 25: Representation of data from Fig. 23 in the direction of the new feature $\varphi(x_1, x_2) = (x_1^2, x_2^2)$.

10.1.1 Representer Theorem

After this introductory part, we formalize the concepts introduced above. We start with a general result, useful for the following. Let us consider $X \in \mathbb{R}^{n \times d}$ and the total loss function:

$$\mathscr{L}(w) = \sum_{\mu=1}^{n} \ell(y_\mu, w \cdot X_\mu) + \lambda \sum_{i=1}^{d} w_i^2 \tag{10.4}$$

We can then state the following important theorem:

Representer theorem: Any minimizer of $\mathscr{L}(w)$ can be written as :

$$\hat{w} = \sum_{\mu=1}^{n} \alpha_\mu X_\mu; \quad \alpha \in \mathbb{R}^n; \quad X_\mu \in \mathbb{R}^d; \quad \mu = 1, ..., n; \quad \hat{w} \in \mathbb{R}^d \tag{10.5}$$

Proof: If $\{X_\mu\}_{\mu=1}^n$ contains a basis, the proof is trivial because by definition of a basis any vector can be written as a linear combination of basis vectors.

If not: any generic vector $w \in \mathbb{R}^d$ can be written

$$\begin{aligned} w = \sum_{\mu=1}^{n} \alpha_\mu X_\mu + v, \text{ such that} \\ v \cdot X_\mu = 0 \ \forall \mu \end{aligned} \tag{10.6}$$

Id est, v is in the null space. Because of that, $l = l(y_\mu, w \cdot X_\mu)$ does not depend on v, so the only dependency of Eq.(10.4) on v remaining is in the w_i^2 term:

$$\begin{aligned} \sum_{i=1}^{d} w_i^2 &= \sum_{i=1}^{d} (\sum_{\mu=1}^{n} \alpha_\mu X_{\mu i} + v_i)^2 \\ &= \sum_{i=1}^{d} (\sum_{\mu=1}^{n} a_\mu X_{\mu i})^2 + \sum_{i=1}^{d} v_i^2 + 2 \sum_{i=1}^{d} v_i \sum_{\mu=1}^{n} \alpha_\mu X_{\mu i} \,. \end{aligned} \tag{10.7}$$

Again because v is in the null space, the double sum is equal to zero, and as such the only term remaining is the v^2 one. Since the two remaining terms in the above equation are squared, each term can be minimized individually. Minimizing this the v containing term is trivial: The minimized v is just $v = 0$. In conclusion:

$$\hat{w} = \sum_{\mu=1}^{n} \alpha_\mu X_\mu, \quad \text{QED.} \tag{10.8}$$

Applying the representer theorem (10.5) on the loss function (10.4) gives:

$$\begin{aligned}\mathscr{L}(\alpha) &= \sum_{\mu=1}^{n} l(y_\mu, \sum_{i=1}^{d}\sum_{\nu=1}^{n} \alpha_\nu X_{\nu i} X_{\mu i}) + \lambda \sum_{i=1}^{d} (\sum_{\nu=1}^{n} \alpha_\nu X_{\nu i})^2 \\ &= \sum_{\mu=1}^{n} l(y_\mu, \sum_{\nu=1}^{n} \alpha_\nu (\sum_{i=1}^{d} X_{\nu i} X_{\mu i})) + \lambda \sum_{\nu,\mu} \alpha_\nu \alpha_\mu \sum_{i=1}^{d} X_{\nu i} X_{\mu i}\,,\end{aligned} \tag{10.9}$$

which naturally leads to the definition of the *Gram Matrix*:

$$K_{\mu\nu} := \sum_{i=1}^{d} X_{\nu i} X_{\mu i} = X_\mu \cdot X_\nu\,. \tag{10.10}$$

10.2 Definition of Kernels

10.2.1 Kernels as an alternative scalar product

Now, instead of minimizing over the weights w, we can try to minimize over the unknown coefficients α_ν. Basically, we are rewriting the linear regression in an equivalent way. A priori there are no particular reasons to minimize over α instead of over w. We can consider some advantages, for example if $n < d$, we are minimizing over n parameters instead of over d, and this leads to a faster linear regression. So, if $n < d$, minimisation over α is faster, otherwise minimisation over w remains the best from a computational time point of view. Given the new data sample X_{new}, we can express the prediction y_{new} in terms of α_ν:

$$y_{\text{new}} = \sum_{i=1}^{d} w_i X_{\text{new,i}} = \sum_{i=1}^{d} \left(\sum_{\nu=1}^{n} \alpha_\nu X_{\nu i} \right) X_{\text{new,i}} = \sum_{\nu=1}^{n} \alpha_\nu \sum_{i=1}^{d} X_{\nu i} X_{\text{new,i}} = \sum_{\nu=1}^{n} \alpha_\nu K_{\nu\text{new}}. \tag{10.11}$$

We define the α that we find through the minimization procedure, $\hat{\alpha} = \text{argmin}_\alpha \mathscr{L}(\alpha)$.

The scalar product $K_{\nu\mu} = \sum_{i=1}^{d} X_{\nu i} X_{\mu i}$ is a measure of the similarity between X_ν and X_μ. For example if the angle between X_ν and X_μ is large, the scalar product is small and the two vectors are not similar, and vice versa if the scalar product is large and the vectors are similar. With this example in mind, we can consider other ways to express similarity between two vectors, not only the scalar product. The loss function can be written in a more general way:

$$\mathscr{L}(\alpha) = \sum_{\mu=1}^{n} \ell \left(y_\mu, \sum_{\nu=1}^{n} \alpha_\nu K(X_\nu, X_\mu) \right) + \lambda \sum_{\nu=1}^{n} \sum_{\mu=1}^{n} \alpha_\nu \alpha_\mu K(X_\nu, X_\mu)\,, \tag{10.12}$$

where we introduced the *kernel* $K(X_\nu, X_\mu) \equiv K_{\nu\mu}$, which is, in our presentation, an alternative to the scalar product, a *measure of similarity*. We can of course define many different kernels, each representing a different way to quantify similarity between X_ν and X_μ:

$$K(X_\nu, X_\mu) = X_\nu \cdot X_\mu, \quad K(X_\nu, X_\mu) = e^{X_\nu \cdot X_\mu}, \quad K(X_\nu, X_\mu) = e^{-||X_\nu - X_\mu||^2}\,, \tag{10.13}$$

where the first is the *scalar product*, the second is the *exponential kernel* and the third is the *radial basis kernel*.

Now we have to implement a procedure for the minimization of $\mathscr{L}(\alpha)$, in order to find the $\hat{\alpha}$ which minimizes the loss function. We exploit the gradient descent with respect to $\alpha \in \mathbb{R}^n$. We note that if ℓ is convex (we remind that a real valued function is convex if the line segment between any two points on the graph of the function does not lie below the graph between the two points), then the minimizer is unique. This procedure is the *kernel regression* and the predictor y_{new} can be written as:

$$y_{\text{new}} = \sum_{\nu=1}^{n} \hat{\alpha}_\nu K(X_\nu, X_{\text{new}})\,, \tag{10.14}$$

which is a linear combination (through the coefficients $\hat{\alpha}$) of similarity of a new data sample to all the previous training data sample. This is the simplest way to introduce kernels, but there is another, more formal way to define them.

10.2.2 Kernels via Feature Maps

Another way to define a kernel involves the concept of a feature map, we remember that a feature map φ takes a d-dimensional vector X_μ and transforms it into a p-dimensional vector $\varphi(X_\mu)$, so $\varphi : \mathbb{R}^d \to \mathbb{R}^p$. We can write the loss function in terms of feature map:

$$\mathscr{L}(w) = \sum_{\mu=1}^{n} \ell \left(y_\mu, \sum_{a=1}^{p} w_a \varphi_a(X_\mu) \right) + \lambda \sum_{a=1}^{p} w_a^2 \,, \tag{10.15}$$

where $\varphi_a(X_\mu)$ appears instead of the simple vector. Basically, we are doing a linear regression in a features space. We recognize that $\hat{w}_a = \text{argmin}_a \mathscr{L}(w)$. Then, we can apply the representer theorem in order to write w_a in terms of the feature map $\varphi_a(X_\mu)$:

$$w_a = \sum_{\mu=1}^{n} \alpha_\mu \varphi_a(X_\mu) \,. \tag{10.16}$$

If we substitute the last expression in $\mathscr{L}(w)$ ((10.15)) we get:

$$L(\alpha) = \sum_{\mu=1}^{n} \ell \left(y_\mu, \sum_{a=1}^{p} \sum_{\nu=1}^{n} \alpha_\nu \varphi_a(X_\mu) \varphi_a(X_\nu) \right) + \lambda \sum_{a=1}^{p} \sum_{\nu=1}^{n} \sum_{\mu=1}^{n} \alpha_\nu \alpha_\mu \varphi_a(X_\mu) \varphi_a(X_\nu) \,. \tag{10.17}$$

If we compare the last expression with the loss (10.12) we recognize the kernel:

$$K(X_\mu, X_\nu) = \sum_{a=1}^{p} \varphi_a(X_\mu) \varphi_a(X_\nu) \tag{10.18}$$

The last expression is the so-called *kernel gram matrix*, which is now expressed through features maps. In particular it is the standard scalar product in the direct space.

The question now is: How can we relate the kernel gram matrix expressed through feature maps with the exponential kernel, with the radial basis kernel, or with another generic kernel? To which feature maps do these kernels correspond?

10.3 Feature Maps for common Kernels

The relationship between Kernel formulation and feature maps is extremely useful when a high (or even infinite) dimensional feature space is considered. In linear regression, the objective is to find a function that correctly generates the outputs from the input data X. It must be general enough to represent all the functions able to perform this transformation from X to Y. In high dimensional space, an example can be drawn from the Taylor series. They approximate with enough generality any function by using series of infinitely many polynomials. However, this requires the feature map to have an infinite dimension $p \to \infty$, thus the computation of the expression in (10.17) is not possible. Under some assumptions, it can be shown that the infinite-dimension feature maps correspond exactly to Kernels, Eq.(10.12), allowing feasible computations. Indeed, considering the Kernel formulation in (10.12), the dimension of the feature space does not appear anymore and the most expensive computation to perform would be over $n \times n$ matrices.

10.3.1 Example in dimension one

To start, the case $d = 1$ is considered, where x is a real number. We consider the following feature map:

$$\varphi(x) = (1, x, x^2, x^3, x^4, ..x^p) \quad \text{with} \quad p \to \infty \tag{10.19}$$

The product from equation (10.18) becomes:

$$\sum_{a=1}^{\infty} \varphi_a(X_\nu) \varphi_a(X_\mu) = \sum_{p=0}^{\infty} X_\nu^p X_\mu^p = \frac{1}{1 - X_\mu X_\nu} \,. \tag{10.20}$$

The scalar product of the feature map is a geometric series and the last equality holds only if $X_\nu X_\mu < 1$. Under this assumption, the scalar product of the infinite-dimension feature map φ considered can be computed by using the Kernel:

$$K(X_\mu, X_\nu) = \frac{1}{1 - X_\mu X_\nu} \,. \tag{10.21}$$

Although the Kernel formulation implicitly represents infinite dimensional spaces, it allows to perform regression over much smaller matrices of $n \times n$ dimensions.

10.3.2 General Dimension

In order to ensure generality of the Kernel formulation of feature maps, we now consider the case of general d dimensions. Consider the previous kernel in general dimension d:

$$K(X_\mu, X_\nu) = \frac{1}{1 - X_\mu \cdot X_\nu} = \sum_{p=0}^{\infty} (X_\mu \cdot X_\nu)^p. \tag{10.22}$$

The term in the summation on the right-hand side can be expanded as a multinomial, such that:

$$\sum_{p=0}^{\infty} (X_\mu \cdot X_\nu)^p = \sum_{p=0}^{\infty} \sum_{\beta_1+\beta_2+\cdots+\beta_d=p} \frac{p!}{\prod_{i=1}^{d} \beta_i!} \prod_{i=1}^{d} (X_{\mu i} X_{\nu i})^{\beta_i}. \tag{10.23}$$

We can note here that the coefficient $\frac{p!}{\prod_{i=1}^{d} \beta_i!}$ corresponds to the number of ways to place p objects in d boxes such that there are β_1 elements in box 1, β_2 elements in box 2 and so on. From this kernel, we find that the feature space can be constructed as:

$$\varphi_{\beta_1,\beta_2,\cdots,\beta_d}(X_\mu) = \sqrt{\frac{p!}{\prod_{i=1}^{d} \beta_i!}} \prod_{i=1}^{d} X_{\mu i}^{\beta_i}. \tag{10.24}$$

We can now treat another example of kernel: the exponential kernel. Proceeding as before we have:

$$K(X_\mu, X_\nu) = e^{X_\mu X_\nu} = \sum_{p=0}^{+\infty} \varphi_p(X_\mu)\varphi_p(X_\nu) = \sum_{p=0}^{+\infty} \frac{(X_\mu \cdot X_\nu)^p}{p!}. \tag{10.25}$$

For this kernel, exploiting the multinomial expansion, the feature space can be constructed as:

$$\varphi_{\beta_1,\beta_2,\cdots,\beta_d}(X_\mu) = \sqrt{\frac{1}{\prod_{i=1}^{d} \beta_i!}} \prod_{i=1}^{d} X_{\mu i}^{\beta_i}. \tag{10.26}$$

So we find that the features are the same as in Eq.(10.24), with only one coefficient changed. Note that in practice, the exponential kernel is used more often for normalised data, in which case $||X_\mu|| = ||X_\nu|| = 1$, which would give $X_\mu \cdot X_\nu = \cos(\theta)$, where θ is the angle between the data.

The final kernel we have introduced in Eq.(10.13), the radial basis kernel, is the most commonly used. Its feature map is given by

$$\Phi_{\beta_1,\beta_2,\cdots,\beta_d}(X_\mu) = \sqrt{\frac{1}{\prod_{i=1}^{d} \beta_i!}} \sqrt{2^p} \exp\left\{ -\sum_{i=1}^{d} {X_{\mu i}}^2 \right\} \prod_{i=1}^{d} X_{\mu i}^{\beta_i} \tag{10.27}$$

and is also called the *radial basis function* (RBF).

10.4 Interpretation and Drawbacks

To interpret what kernel regression does what, it is illustrative to consider a one-dimensional example. Try to visualise the action of the following function;

$$f(X) = \sum_{\nu=1}^{n} \hat{\alpha}_\nu K(X_\nu, X) \tag{10.28}$$

where we learn some coefficients $\hat{\alpha}_\nu$ from the optimisation problem (10.12). For $n = 5$ and using the radial basis function (RBF) kernel, the function $f(X)$ is Gaussian around each of the five training points X_ν. Each term in the sum concerns one of the training points, and the function shape is given by the kernel (each amplitude depends on $\hat{\alpha}_\nu$). The further from the training point, the less is added because the kernel is just a similarity (here a Gaussian).

We then add the data. For example, if we have the X_{new} as shown on Figure 26 and compute the $f(X)$ corresponding to it, we basically do not count contributions from X_1 or X_5 as they have little weight at X_{new}, but we average the contributions from X_3, X_4, and X_2 because of their coefficients and their distance (similarity) to X_{new}. This is similar to the k-nearest neighbours method – the kernel method computes a *weighted* average of the training points that are nearby in terms of $K(X_\nu, X_\mu)$. Unfortunately, this means the method may suffer from the curse of dimensionality in the same way and for the same reasons as the k-nearest neighbours did (but in general this problem is slightly mitigated because of the weights).

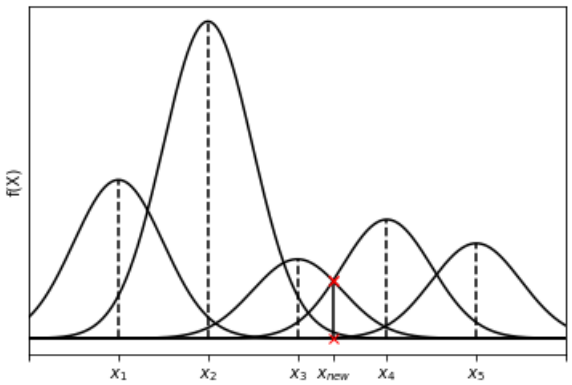


Figure 26: Function $f(x)$ around five different X_i's.

Another possible drawback is more computational. Since we could not work in infinite-dimensional space, instead for kernel regression we worked in n-dimensional space. Now $K(X_\mu, X_\nu) \in \mathbb{R}^{n\times n}$; for $n \sim 10^3$ this is fine, but for modern data sets with $n \sim 10^6$ or higher, we struggle to store $n \times n$ matrices this large and doing computations with them seems intractable. We would like to do the same thing but more computationally efficient – *random features* are a way to approximate kernels which can save us from $n \times n$ matrices that are too large. In the next lecture we use random features as a way to derive neural networks.

11 Lecture 11: Neural networks 3.12.

Students designated to write this section: Falch Håvard, Ernst Samuel, Dufour Charles François Paul Pierre, Delévaux David, De Schoulepnikoff Paulin Jean, De Lucca Brenno Jason Sanzio Peter
Coordinator:
TA: Alessandro (TA approved)
Students who actually contributed: Ernst Samuel, Dufour Charles François Paul Pierre, Håvard Falch, De Schoulepnikoff Paulin Jean, Delévaux David, De Lucca Brenno Jason Sanzio Peter

11.1 Reminders on Kernels

Let us consider the data matrix $X \in \mathbb{R}^{n\times d}$, with labels $y \in \mathbb{R}^n$, and p features $\phi_a(X_\mu) \in \mathbb{R}$, with $a = 1, \ldots, p$. Regression with features consists in minimizing the loss function:

$$\mathscr{L}(w) = \frac{1}{n}\sum_{\mu=1}^{n} l\Big(y_\mu; \sum_{a=1}^{p} w_a\phi_a(X_\mu)\Big) + \lambda\sum_{a=1}^{p} w_a^2 \tag{11.1}$$

with respect to w.

In Lecture 10, we saw that this is equivalent to *kernel regression*, which consists in minimizing the loss function

$$\mathscr{L}(\alpha) = \frac{1}{n}\sum_{\mu=1}^{n} l\Big(y_\mu; \sum_{\nu=1}^{n} \alpha_\nu K(X_\mu, X_\nu)\Big) + \lambda\sum_{\mu,\nu} \alpha_\mu\alpha_\nu K(X_\mu, X_\nu) \tag{11.2}$$

with respect to α. The *Kernel Gram matrix* $K(X_\mu, X_\nu) = \sum_{a=1}^{p} \phi_a(X_\mu)\phi_a(X_\nu)$ is a scalar product in the feature space. For some features, this leads to simple functions such as the radial basis kernel $K(X_\mu, X_\nu) = e^{-||X_\mu - X_\nu||^2}$, which is one of the most commonly used kernels.

Support vector machine (SVM) is a kernel method for classification, where the loss function is the hinge loss function (see section 5.1):

$$l(y_\mu, z_\mu) = \max(0, 1 - y_\mu z_\mu)\,. \tag{11.3}$$

For linear classification, we had $z_\mu = w \cdot X_\mu = \sum_{i=1}^{d} w_i X_{\mu i}$. Here, in SVM, we have $z_\mu = \sum_{\nu=1}^{n} \alpha_\nu K(X_\nu, X_\mu)$.

11.2 Random feature regression

As we saw, the Kernel Gram matrix $K \in \mathbb{R}^{n\times n}$ has matrix elements $K_{\mu\nu} = K(X_\mu, X_\nu)$. Computing this matrix may be impractical if n is very large. In order to mitigate this issue, we introduce *random feature regression*. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be some non-linear function, e.g., $\sigma(z) = \text{sign}(z)$. One can take D random vectors $F_a \in \mathbb{R}^d$, e.g., from the distribution:

$$P(F_a) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}F_{ai}^2}, \quad \text{for} \quad a = 1, ..., D, \tag{11.4}$$

from which we can introduce D features as:[19]

$$\Phi_a(X_\mu) = \sigma(F_a \cdot X_\mu), \quad \text{for} \quad a = 1, ..., D. \tag{11.5}$$

At a first glance, it may not be obvious why this is an interesting or even a good idea. Indeed, all we did was taking random linear combinations of the data and applying a non-linear function. To understand the idea behind the approach, let us take the limiting case where $D \to \infty$. A key property is that, in this limit, we have:

$$\sum_{a=1}^{\infty} \Phi_a(X_\mu)\Phi_a(X_\nu) = \mathbb{E}_{P(F)}\left[\sigma(F \cdot X_\mu)\sigma(F \cdot X_\nu)\right] = K(X_\mu, X_\nu) \tag{11.6}$$

for some Kernel $K(X_\mu, X_\nu)$. Thus, taking infinitely many features, we find a direct relation between kernels K and $(P(F), \sigma)$. In fact, random feature regression is equivalent to kernel regression as $D \to \infty$. However, we hope that under truncation, i.e., taking D finite but sufficiently large, this advantage of kernel methods will pertain. Based on our new features (11.5) and the property (11.6) let us define the loss function:

$$\mathscr{L}(w) = \frac{1}{n}\sum_{\mu=1}^{n} l\left(y_\mu, \sum_{a=1}^{D} w_a \sigma\left(\sum_{i=1}^{d} F_{ai}X_{\mu i}\right)\right) + \lambda \sum_{a=1}^{D} w_a^2, \quad \text{for large } D, \tag{11.7}$$

which is trainable as before by minimizing the parameters w. Now, if n is very large, we can choose D smaller to regain computational feasibility. Typically, we choose $d < D < n$.

To summarize, random feature models have the advantages of the kernel regression but may be much faster to compute for a carefully chosen D.

11.3 Neural Networks

The main idea of neural networks is to see the random features used in random feature regression Eq.(11.6) as learnable parameters. Indeed, the features F will not be random anymore but learned (e.g., with stochastic gradient descent) together with the weights w. Our regression problem can then be rewritten as:

$$\begin{cases} \min\limits_{w,F} \mathscr{L}(w, F) \\ \mathscr{L}(w, F) = \frac{1}{n}\sum\limits_{\mu=1}^{n} l\left(y_\mu, \sum\limits_{a=1}^{D} w_a \sigma\left(\sum\limits_{i=1}^{d} F_{ai}X_{\mu i}\right)\right) \end{cases} \tag{11.8}$$

and we will predict using:

$$y_{\text{new}} = \sum_{a=1}^{D} w_a \sigma\left(F_a \cdot X_{\text{new}}\right), \tag{11.9}$$

where σ is a non-linear function[20] that needs to be specified. In the above expression, we did not include the *bias* term in order to simplify the notation, but one can generalise easily to that case by simply adding a column of 1's to the input (as we did in the case of linear regression).[21] We can represent graphically Eq.(11.9) using Figure 27.

[19] Note that σ acts component wise.

[20] If σ were a linear function, the whole model would boil down to a linear one.

[21] Instead of adding a column to the input it might be intuitively easier think of the bias added as:

$$\sigma(F_a \cdot X_{\text{new}}) \longrightarrow \sigma(F_a \cdot X_{\text{new}} + b_a). \tag{11.10}$$

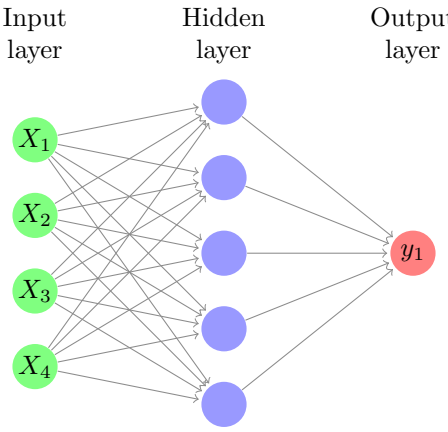


Figure 27: 1-hidden layer feed forward fully connected neural network with $D = 5$ and $d = 4$. To make the link with the Equations (11.8)) and (11.9)), the weight of the arrows connecting the input layer to the hidden layer (respectively the hidden layer to the out-layer) would correspond to the vector F (respectively w). It's also called a 2 layers network given that there are two vectors to minimize, w and F.

This function is called a one hidden layer feed forward neural network where:

- $X_\mu \in \mathbb{R}^d$ is the input,
- $F \in \mathbb{R}^{D\times d}$ are the weights of the first layer,
- D is the width of the hidden layer,
- σ is the activation function,
- $w \in \mathbb{R}^D$ are the weights of the second layer,
- y_μ is the label of sample μ,
- $h_{a\mu} = F_a \cdot X_\mu$ is the pre-activation of hidden neuron a and sample μ,
- $t_{a\mu} = \sigma(F_a \cdot X_\mu) = \sigma(h_{a\mu})$ is the post-activation of hidden neuron a and sample μ.

Note that we can write the generalized linear regression model as a neural network. It consists of a one-layer neural network (no hidden layer).

11.3.1 Universal Approximation Theorem

We can write the class of functions representable by a one-hidden layer neural network from $\mathbb{R}^d$ to $\mathbb{R}$ as:

$$\mathscr{F}_{W,F,b} := \left\{ f_{W,F,b}(X) = W^T \sigma(FX + b) \text{ for } F \in \mathbb{R}^{D\times d}, b \in \mathbb{R}^D, W \in \mathbb{R}^D \right\}, \tag{11.11}$$

where sometimes we omit the bias b to simplify the notation, which yields:

$$\mathscr{F}_{W,F} := \left\{ f_{W,F}(X) = W^T \sigma(FX) \text{ for } F \in \mathbb{R}^{D\times d}, W \in \mathbb{R}^D \right\}.$$

The universal approximation theorem[22] states that any continuous bounded target function $f^* : \mathbb{R}^d \mapsto \mathbb{R}$ can be approximated by a *1-hidden layer feed forward fully connected neural network* with an activation function σ that is not a polynomial function, such that $\forall \varepsilon > 0$, there exists a $f_{W,F} \in \mathscr{F}_{W,F}$ with

$$\sup_{X\in\mathbb{R}^d} \|f^*(X) - f_{W,F}(X)\| < \varepsilon. \tag{11.12}$$

This theorem only states the existence of a good network that allows us to approximate any continuous function f^* as precisely as we want, but it does not imply that it can be found efficiently. The optimization problem for neural network is non-convex, so in general we do not control how the gradient descent process behaves: i.e., we can end up in a local minimum. Moreover, the width D may need to be large (exponential in d).

[22] See for example [16] for more details.

11.3.2 Activation Functions

How to choose the right activation function σ? First, σ needs to be a non-polynomial function if we want to apply the universal approximation theorem. In fact, taking for example $\sigma(z) = z$, we would fall back to a linear model. On the other hand, if $\sigma(z) = z^p$ with $p > 1$, the activation function is not generic enough to fit any function.

The most commonly used activation function is called **rectified linear unit** (ReLU):

$$\sigma(x) = \max(0, x). \tag{11.13}$$

Other possible options for the activation function are:

- **hyperbolic tangent**: $\sigma(x) = \tanh(x)$
- **sigmoid function**: $\sigma(x) = 1/(1 + e^{-x})$
- **threshold function**: $\sigma(x) = \theta(x - \kappa)$ for some $\kappa \in \mathbb{R}$.

A clear relation with biological neural networks can be found if we choose the threshold function as the activation function. In this case we have:

$$\theta(\sum_{i=1}^{d} F_{ai} X_{\mu i} - \kappa) = X_{\mu a} \tag{11.14}$$

where F_{ai} can be seen as the synaptic strength from neuron i to neuron a: If $F_{ai} > 0$ it is called an excitatory connection and if $F_{ai} < 0$ an inhibitory one. $X_{\mu i} \in \{0, 1\}$ is the activity of the neuron i at some time μ and $X_{\mu a}$ the activity of neuron a. If the sum of the electric signals $\sum_{i=1}^{d} F_{ai} X_{\mu i}$ coming from the neurons i connected to neuron a exceed the threshold κ, then $X_a = 1$ and neuron a is activated (see Figure 28). The connection between artificial and biological neural networks is a field on its own and is exposed in detail in another course.[23]

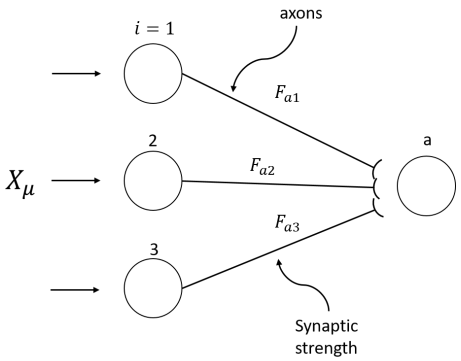


Figure 28: Relation between neural networks and biological neural networks choosing the threshold function as the activation function.

11.4 Multi Layer Neural Networks

A one hidden layer neural network learns one layer of features; however, this can be generalized to a learning the features of the features etc. scheme, which is what a multi-layer neural network does. The idea behind this is to avoid the large dimension D in a single layer neural network as it could possibly be very large. Also, it could be a good idea to learn features of features. In Figure 29 we have a feed-forward fully connected neural network with 4 layers of weights and 3 hidden layers.

The function of a multi-layer neural network is:

$$f_W(X_\mu) = \phi^{(L)}\left(\sum_{a_L}^{p_L} W^{(L)} \cdots \phi^{(2)}\left(\sum_{a_2}^{p_2} W^{(2)}_{a_3 a_2} \phi^{(1)}\left(\sum_{a_1}^{p_1} W^{(1)}_{a_2 a_1} X_{\mu a_1}\right)\right)\right), \tag{11.15}$$

[23] Look at the spring course *Biological modelling of neural networks* (BIO-465) by Prof. Gerstner.

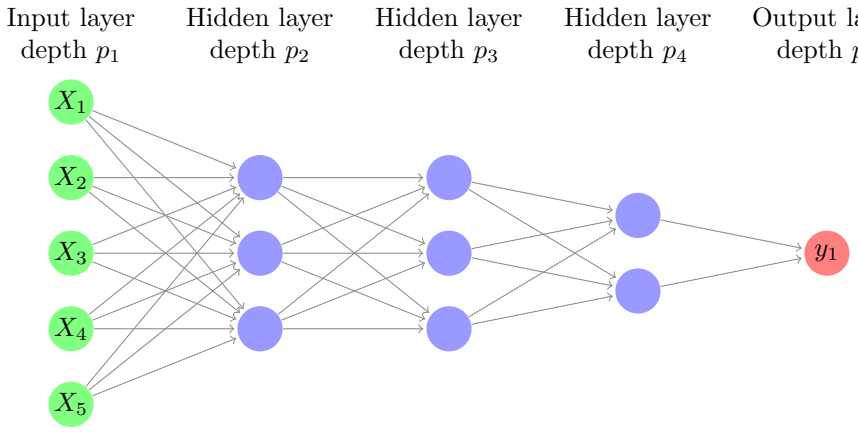


Figure 29: 3-hidden/4-layer feed forward fully connected neural network.

where $X_\mu \in \mathbb{R}^d$ is the input, W denotes the collection of the weights (it would have included F and w in the previous case), $\phi^{(l)}$ is the activation function[24] of layer l, L is the number of layers, or depth, and p_l is the width of layer l where $p_1 = d$ and $p_{L+1} = 1$. In matrix notation this becomes:

$$f_W(X_\mu) = \phi^{(L)}(W^{(L)} \cdots \phi^{(2)}(W^{(2)}\phi^{(1)}(W^{(1)}X_\mu))) \tag{11.16}$$

where the activation functions act component wise.

For training of a multi-layer neural network, we do the same as for one-hidden-layer neural networks

$$\mathscr{L}(\{W^{(l)}\}_{l=1}^{L}) \equiv \mathscr{L}(W) = \sum_{\mu=1}^{n} l(y_\mu; f_W(X_\mu)) \tag{11.17}$$

To minimize this loss function, we use Stochastic Gradient Descent (SGD) with respect to W, even though the loss function is highly non-convex.

12 Lecture 12: Deep neural networks 10.12.

Students designated to write this section: Dardano Thomas, Cucurachi Daniele, Claudon Baptiste Paul François, Carranza I Botey Ester, Brown Steven Sinclair, Briand Guillaume Miro André
Coordinator: Guillaume Briand
TA: Federica (TA approved)
Students who actually contributed: Dardano Thomas, Cucurachi Daniele, Claudon Baptiste Paul François, Carranza I Botey Ester, Brown Steven Sinclair, Briand Guillaume Miro André

12.1 Recap on Multilayer Neural Network / Deep Learning

Let's consider a *forward fully-connected neural network* (i.e., each neuron of a certain layer is connected to the all neurons of the successive layer), where we call L the number of layers, $X \in \mathbb{R}^{n\times d}$ the inputs and $y \in \mathbb{R}^n$ the labels, with n the number of samples, and d the input dimension. We also define p_l as the number of neurons in the l^{th} layer, with $l = 1, ..., L$. When considering regression problems, we have that the size of the output layer, namely p_L is equal to 1, while we can have $p_L = k$ in the case of classification problems, where k is the number of classes.

The connections among the layers encode for the weights, which are the matrices $W^{(l)}$. The network in Figure 30 represents in practise a function $f : \mathbb{R}^d \to \mathbb{R}^{p_{L+1}}$, such that:

$$f_{W=\{W^{(l)}\}_{l=1}^{3}} = \varphi^{(3)}\left(\sum_{a_3=1}^{p_3} W_{a_3}^{(3)}\varphi^{(2)}\left(\sum_{a_2=1}^{p_2} W_{a_3a_2}^{(2)}\left(\varphi^{(1)}\left(\sum_{i=1}^{d} W_{a_2a_1}^{(1)}X_{\mu i}\right)\right)\right)\right), \tag{12.1}$$

[24] Note that it is a question of definition whether σ, ϕ or φ is used to denote the activation function.

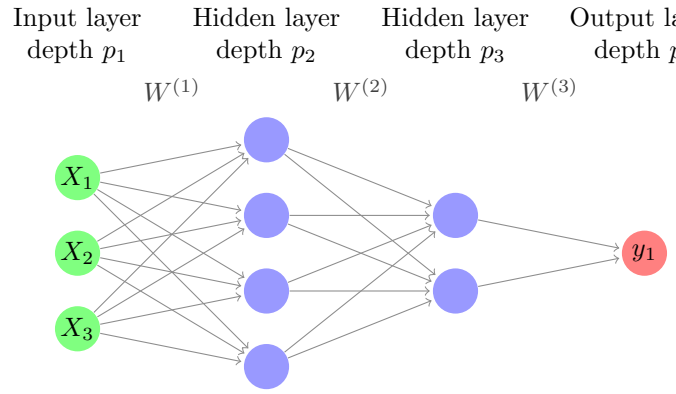


Figure 30: Fully connected neural network with L=3 (number of layers) and $p_1 = 3, \quad p_2 = 4, \quad p_3 = 2, \quad p_4 = 1$. The green bullets correspond to the input neurons, the violet bullets to the hidden units, the red bullet to the output neuron. The black arrows correspond to the weights connecting each neuron of the previous layer to the ones of the subsequent layer.

where:

$$\varphi^{(l)} \qquad \text{is the activation function}$$

$$a_1 = 3 \qquad a_2 = 1, ..., p_2 \qquad a_3 = 1, ..., p_3.$$

In each layer, we thus take the scalar product between the inputs to a given layer and the weights connecting this layer to the subsequent one. The scalar product then goes through a non-linearity $\varphi^{(l)}$. Note that in case of a network with a larger number of layers we have the same structure:

$$f_{W=\{W^{(l)}\}_{l=1}^{L}} = \varphi^{(L)}(\sum_{L=1}^{p_L} W^{(L)}_{a_{L+1}a_L} \quad ... \quad \varphi^{(2)}(\sum_{a_2=1}^{p_2} W^{(2)}_{a_3 a_2}(\varphi^{(1)}(\sum_{i=1}^{d} W^{(1)}_{a_2 i} X_{\mu i})))) \,. \tag{12.2}$$

Note that $a_1 = i$ and also that we are omitting the *biases* here: in general $W^{(l)}\varphi^{(l-1)}(\cdot)$ should have also a vector that represents the shifts $W^{(l)}\varphi^{(l-1)}(\cdot) + b^{(l-1)}$. In this lecture the biases $b^{(l-1)}$ are set equal to zero for simplicity.

12.2 Training of a Neural Network

How to train a neural network? The training of neural networks has a lot of similarities with the linear regression: The dataset is split into a training, validation, and test set. On the training set, one minimises the loss. On the validation set, one evaluates the hyperparameters and on the test set, one checks the performance of the model. For neural networks, the loss is expressed as:

$$\mathscr{L}(\{W^{(l)}\}_{l=1}^{L}) = \sum_{\mu=1}^{n} l(y_\mu, f_W(X_\mu)) + \underbrace{(\text{weight decay})}_{\text{omitted in this lecture}} \tag{12.3}$$

where a regularisation, called weight decay in neural network, could be added. However, this is omitted in this lecture as weight decay is often not so essential. To train the neural network, one minimises the loss function $\mathscr{L}$ over $\{W^{(l)}\}_{l=1}^{L}$ using SGD.

12.2.1 Common Losses

Regression: A frequently used loss for regression is the **quadratic loss**, which is given by:

$$l(y, z) = (y - z)^2 \,, \tag{12.4}$$

where y is a label and z its prediction.

Classification: In classification problems, the labels y are one-hot encoded, by a k-dimensional vector holding 0 or 1 as components. The component equal to 1 encodes for the class which the sample belongs to. For instance, a label could be:

$$y_\mu = (0,1,0,0,0)^{\mathrm{T}}, \qquad k = 5, \tag{12.5}$$

meaning that the input X_μ belongs to the class number 2 as the second component of y_μ holds the 1.
A very commonly used loss for classification is the **cross-entropy loss** given by:

$$l(y_\mu, f_W(X_\mu)) = -\sum_{a=1}^{k} y_{\mu a} \ln(p_{\mu a}) \tag{12.6}$$

where y_μ is a one-hot encoded label and $p_{\mu a}$ is given by the *soft-max function*:

$$p_{\mu a} = \frac{e^{h^{(L)}_{a\mu}}}{\sum_{b=1}^{k} e^{h^{(L)}_{b\mu}}} \tag{12.7}$$

with:

$$h^{(L)}_{a\mu} = \sum_{a_L} W^{(L)}_{a a_L} t^{(L-1)}_{a_L \mu}\,. \tag{12.8}$$

where we have summed over a_L. $W^{(L)}_{a a_L}$ denotes the component (a, a_L) of the last layer and $t^{(L-1)}_{a_L \mu}$ the post-activation function of the hidden neuron a_L of sample μ of the $(L-1)$ layer.
To better understand (12.8) let us introduce the **pre-activations**:

$$h^{(l)}_{a_{l+1}\mu} = \sum_{a_l} W^{(l)}_{a_{l+1} a_l} t^{(l-1)}_{a_l \mu} \tag{12.9}$$

where l is the layer's index and μ is the sample's index. Basically $h^{(l)}$ is what one feeds to the activation function $\varphi^{(l)}$, see Eq.(12.2).
On the other hand, the **post-activations**:

$$t^{(l-1)}_{a_l \mu} = \varphi^{(l-1)}(h^{(l-1)}_{a_l \mu}), \tag{12.10}$$

is what comes out from the activation function of the previous layer.

We therefore have the entire picture of what was done in Eq.(12.6). We summed over all classes the product of the one-hot encoded vector and the logarithm of the output of the soft-max function (12.7). The soft-max itself is a function of the pre-activation (12.8) of the last layer. Since we often have multiple layers in neural networks it is clear that the pre-activation of layer (l) is dependent on the post-activation (12.10) of layer $(l-1)$.

Note the analogy of the softmax (12.7) with the probabilities we had in logistic regression (cf. Lecture 5):

$$p_{\mu a} = \frac{e^{X_\mu w_a}}{\sum_{b=1}^{k} e^{X_\mu w_b}} \tag{12.11}$$

The difference with respect to logistic regression is the fact that in logistic regression we have directly the input, whereas in neural networks we have the pre-activations of the last layer of the neural network.

Then, we can proceed to minimize the loss function using the Stochastic Gradient Descent (SGD):

$$W^{(l)}_{ab}(t+1) = W^{(l)}_{ab}(t) - \gamma \sum_{\mu \in B_t} \left. \frac{\partial \ell(y_\mu, f_W(X_\mu))}{\partial W^{(l)}_{ab}} \right|_{W^{(l)}_{ab}(t)} \qquad \forall\, l, ab \tag{12.12}$$

where γ is the *learning rate*, also called *step size*, and B_t is the batch of samples at iteration t.

We will count the time in *epochs*; one epoch is equal to one pass through the whole training set. At each epoch one typically performs a reshuffling of both the order in which the batches are presented, and the samples contained within each batch.

Evaluation

We aim to evaluate the learning performances of the network throughout the training. Therefore, we typically monitor the following quantities:

- *Training loss* and *test loss*;
- *Training error* and *test error*: some measure of the distance between the true label and the prediction of the network, such as the mean squared error between the two. But the error does not necessarily have to be related to the loss.

When evaluating the performance of the network, measures of *accuracy* or *error* are often plotted – as introduced above. For regression and classification tasks, we typically consider:

- **Regression:** Mean-square error without the regularization:

$$\text{MSE} = \frac{1}{n}\sum_{\mu=1}^{n}(y_\mu - f_W(X_\mu))^2 \tag{12.13}$$

- **Classification:** % of correctly predicted examples:

$$A = \frac{1}{n}\sum_{\mu=1}^{n}\delta_{y_\mu,\hat{y}_\mu} \quad \text{where} \quad \hat{y}_\mu = \underset{a}{\text{argmax}}[f_W(X_\mu)]_a \tag{12.14}$$

The reason why we are taking the *argmax* function is because the *soft-max* defines the probability of each example X_μ to belong to class k. Hence, since the labels are encoded in one-hot vectors, the index of the one-hot vector pointing to the maximum score would correspond to the class predicted by the network.

Another point worth noting is that, in those classification tasks where there are lots of classes involved (for instance $K = 100$ classes), one typically looks at the top-k predictions (for example $k = 5$) and check whether the true labels actually correspond to one of the top-k predictions. As a result, this will allow us to define the **top-k accuracy**.

12.2.2 Back-Propagation Algorithm

Now that we can express the loss function as a function of the weights, how can we efficiently optimise them using the SGD algorithm? We need the derivatives of this seemingly complicated function. Fortunately, neural networks are nothing else than a composition of activation functions and affine transformations. Thus, recursively applying the chain rule, we can efficiently compute these gradients. This procedure is described in the following *back-propagation algorithm*.

0. **Initialisation:**

   The back-propagation algorithm starts with a random initialisation of the weights $W^{(l)}$, for each layer.

The main part of the algorithm then consists in passing forth and back through the neural network to evaluate the function, and its derivatives. These stages are described in the following three steps:

1. **Forward pass:**

   For each layer $l = 1, ..., L$, μ the sample in the sampled batch of indices, and starting from $t_\mu^{(0)} \equiv X_\mu$ up to the output $t^{(L)}$, compute the pre-activation $h^{(l)}$ and post-activation $t^{(l)}$:

$$\begin{aligned} h^{(l)}_{a_{l+1}\mu} &= \sum_{a_l} W^{(l)}_{a_{l+1}a_l} t^{(l-1)}_{a_l\mu} \\ t^{(l-1)}_{a_l\mu} &= \varphi^{(l-1)}(h^{(l-1)}_{a_l\mu}), \end{aligned} \tag{12.15}$$

   Since the input signal propagates forward across the layers up to the output, this step is called the forward pass.

2. **Backward pass**

   For each sample μ in the sampled batch of indices, proceed in the following manner. Compute the error term $e_\mu^{(L)}$:

$$e_\mu^{(L)} \equiv \frac{\partial}{\partial t_\mu^{(L)}} \ell(y_\mu, t_\mu^{(L)}) \varphi^{(L)'}(h_\mu^{(L)})$$

   Then, for each $l = L-1, \ldots, 1$ and $a_l = 1, \ldots, P_l$, compute the error terms $e_{a_l,\mu}^{(l-1)}$:

$$e_{a_l,\mu}^{(l-1)} \equiv \varphi^{(l-1)'}(h_{a_l,\mu}^{(l-1)}) \sum_{a_{l+1}=1}^{p_{l+1}} W_{a_{l+1},a_l} e_{\mu,a_{l+1}}^{(l)} \tag{12.16}$$

   This time, the error propagates backward, from the last layer to the previous layer. Thus, it is called the backward pass.

3. **Weight update**

   Once we have backwardly propagated the errors, we can then update the desired weights through gradient descent. By applying the chain rule, one can show that the gradient of the loss in Eq.(12.12) acquires this simple form:

$$W_{ab}^{(l)}(t+1) = W_{ab}^{(l)}(t) - \gamma \sum_{\mu \in B_t} e_{a\mu}^{(l)} t_{b\mu}^{(l-1)}$$

   where γ is the learning rate and B_t is the batch of samples at iteration t.

4. **Repeat**

   Repeat steps 1, 2 and 3 until a stopping criterion is met. For instance, the train error is small enough, or the parameters are not moving anymore, that is arriving at convergence. Or, we can stop at the point where the validation loss or the validation error starts to increase, that is performing early stopping.

12.2.3 Hyperparameters

In Deep neural networks, there is a large number of hyperparameters that can be tuned in order to find the possible best set of weights. These are:

- The number of layers L and the width of each, p_l with $l = 1, ..., L$
- The activation function chosen (ReLU, sigmoid, hyperbolic tangent, etc.)
- The loss function used (regression : $l(y,z) = (y-z)^2$, classification : cross-entropy, etc.)
- The minibatch, B_μ, size and the sampling for the minimization using Gradient Descent.
- The initialization of the weights
- The regularization performed
- The type of Gradient Descent with its stopping criteria (number of epochs, etc.) and learning rate. In general, there are several variants of GD. For instance, momentum which adds information about the weight at previous iterations or many others like Adagrad, Adam, etc.

There is almost no guiding theory to choose these hyperparameters. It is a trial-and-error procedure, build on experience where we typically tune an hyperparameter and analyse the effect on a specific dataset.

12.2.4 Historical Note

Multi-layer neural networks and their training algorithm aroused great interest in the 90s. Nevertheless, they did not work as well as other methods (like kernel methods for example). More than just the careful fine-tuning of hyperparameters, two things have helped neural networks become much more interesting and revolutionised machine learning:

1. GPUs (Graphical Processing Units): Due to the weak parallel computation capability of CPUs, GPUs are the preferred processing units. Indeed, GPUs have massive parallel computing power and can-do simple calculation, like the ones required in back-propagation, on a very large scale and in reasonable time.

2. Big Data: Clean and large datasets of labelled images are recent:

   - MNIST, the handwritten digits database with $\approx$ 70k digits, developed in 1998.
   - CIFAR-10 (CIFAR-100), from the Canadian Institute For Advanced Research developed in 2009: The dataset contains $\approx$ 60k colored images in 10 (100) different classes (dogs, cats, etc.).
   - ImageNet in 2010, with a dataset of 14 million images for 1000 classes. Thanks to Amazon Mechanical Turk, a crowd-sourcing platform where people around the world are paid to label images, researchers have been able to obtain a large database to train their neural networks.

In 2012, *Deep Learning* increased the accuracy on ImageNet from $\approx$ 74 % to 85 %, a jump never observed before in the Machine learning domain. It is often seen as the official start of the Deep Learning revolution.

12.3 Convolutional Neural Networks (CNN)

Motivation

Fully connected networks can classify with high accuracy the image dataset of MNIST. However, this performance drops for CIFAR-10 and completely falls for ImageNeT. The natural question to ask at this point is: What is missing in a fully connected neural network to that they would work more accurately on images? There are two ingredients missing:

- **Locality:** By construction, a fully connected neural network cannot resolve locality. This is due to the fact that $f_W(X_\mu)$, cf. Eq.(12.1), is invariant under permutation of $X_{\mu i}$, where the input $X_{\mu i}$ would correspond to a pixel. This means that the fully connected neural network cannot distinguish between an image and one of its permutations. However, when humans analyse, for example, a picture of a cat, they can notice that the eyes are situated next to each other. This is one element that allows identification. If we were to take instead, a randomly permutated image of the cat, the human would almost certainly not be able to recognise that the picture depicts a cat. So, locality is a key factor in image recognition.

- **Translational invariance:** If the cat in the image is shifted along a given direction, for a human this would not change anything for identifying it. This translational invariance should also be implemented in the neural network for image analysis.

This is the motivation leading to introduce *convolutional neural networks* (CNN), which is a class of neural networks that includes locality and translational invariance.

Types of Layers

Convolution Neural Networks (CNN) are similar to ordinary neural networks but use the assumption that the inputs are images ($\rightarrow$ locality and translational invariance). There are 3 types of layers in CNNs:

- fully connected layers
- pooling layers
- convolutional layers

12.3.1 Pooling Layers

Let us introduce in this lecture the concept of *pooling* (or down-sampling) layers before discussing in lecture 13 the convolutional layers. Pooling is a non-linear operation that is applied to a layer. It filters the layer by using a so-called pooling function that summarizes nearby inputs. Pooling reduces dimensionality and helps to make the input invariant to translations. Some typical pooling functions are:

- Max pooling: Max pooling constructs a new layer by applying a filter with fixed dimensions, for example 2x2. Note that theses dimensions would be added to the list of hyperparameters that need to be tuned. The matrix of input pixels (or post-activation in the hidden layers) is divided into patches of the same filter size. A lower dimensional matrix is then constructed by choosing the maximum value in each patch. An example is illustrated in Figure 31.

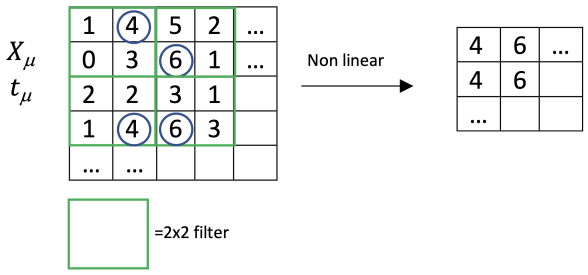


Figure 31: Example of max pooling with 2x2 filter applied to an image input or post-activation.

- Average pooling: The average pooling is the analogous operation of max pooling, where instead of taking the maximum of each patch, it performs an average over all elements in a given patch.

13 Lecture 13: Convolutional networks and deep learning modus operandi 17.12.

Students designated to write this section: Brenna Greta Yin-Sang, Boucard Manon Nathalie Laura, Bolognini Gaia Stella, Bianchi Yannick, Bergerioux Hugo Georges Henri, Berdyev Aman
Coordinator:
TA: Giovanni (TA approved)
Students who actually contributed:Brenna Greta Yin-Sang, Boucard Manon Nathalie Laura, Bolognini Gaia Stella, Bianchi Yannick, Bergerioux Hugo Georges Henri

13.1 Convolutional Neural Networks (CNNs)

13.1.1 Recap on motivations and ideas

For a fully connected neural network, one has the following output:

$$f_W(X_\mu) = \varphi^{(L)}\left(\sum_{a_L=1}^{p_L} W_{a_L}^{(L)} \cdots \varphi^{(2)}\left(\sum_{a_2=1}^{p_2} W_{a_3 a_2}^{(2)} \varphi^{(1)}\left(\sum_{i=1}^{d} W_{a_2 i}^{(1)} X_{\mu i}\right)\right) ..\right) \tag{13.1}$$

with the post activations:

$$t_{a_l}^{(l-1)\mu} = \varphi^{(l-1)}\left(\sum_{a_{l-1}=1}^{p_{l-1}} W_{a_l a_{l-1}}^{(l-1)} t_{a_{l-1}\mu}^{(l-2)}\right) . \tag{13.2}$$

This function has the inconvenient property to be invariant to pixel permutations. This means that if two networks are trained with the same data, but only differ by a pixel permutation, the final performance (in terms of test error) will be the same. This is an issue as it does not exploit the *local structure* of the data, and therefore is not well-matched to it. The usage of *convolutional neural networks* solves this as it allows exploiting this structure. The different layers introduced at the end of lecture 12 of a convolutional neural network are more in detail:

- **Fully connected layers:** As defined in the previous lecture.
- **Pooling layer:** This is a layer where we reduce the dimension of a layer by combining multiple neuron outputs in a pool (also called patch), using a function providing the maximum or the average of this pool, as we can see in the Figure 32. In the beginning there are 36 neurons in the previous layer, by applying the pooling layer 4 neighbouring neurons are gathered in a pool (each of the red squares on the left), finally by taking the maximum or the average, the pool is reduced to 1 red neuron – a new layer is formed. At the end there are 9 neurons. This has the effect of reducing the dimensionality of the previous layer. This layer also acts in the case of max-pooling as a non-linearity, similar to the activation function of the fully connected layer.

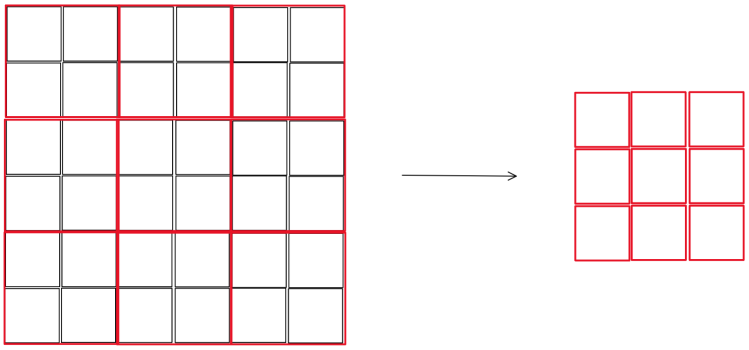


Figure 32: Illustration of the pooling layers

- **Convolutional layers**: They are defined with the following post-activation function:

$$t_{a,\alpha}^{(l+1),\mu} = \varphi^{(l)} \left(\sum_{\beta=1}^{C_l} \sum_{b=1}^{w} W_{\alpha,\beta,b}^{(l)} t_{r_a(b),\beta}^{(l),\mu} \right), \tag{13.3}$$

where a is the *position* in the convolution layer, w the *filter size*, α the *index of the channel* of the convolutional layer, l the layer, μ the sample and C_l the *number of channels* in the layer l. $r_a(b)$ is the b^{th} neuron in the *receptive field* of filter a.

We take as an example the Figure 33, where the input to the layer is one-dimensional to simplify the illustration. The first block (on the left) is the image, one column corresponds to the one-dimensional row of pixels of the image. In this example we consider an image made of three channels (typically red, blue, green). The second block (on the right) is the convolutional layer. Let us take the first column of the output of this convolutional layer, each neuron in this channel looks at neurons in its receptive field, which is a subset of the neurons on the input to the layer, across all channels, as marked in Figure 33.

The size of the receptive field is defined by the size of the filter (number of neurons in the receptive field of the filter), for our case $w = 4$. A *stride* of 2, here means that each convolutional neuron looks at a receptive field of the same size and shape but shifted from the previous one by 2. It is important to note that each convolutional neuron of the considered channel share the same weights $W_{\alpha,\beta,b}^{(l)}$, i.e., this terms does not depend on the index a. This is known as weight sharing; therefore they all share the same filter that is convoluted on the entire image. We can choose several filters which are reflected as several channels/columns on the right block of the Figure 33.

Compare the expression (13.3) for the convolutional layer to the one of fully connected layer (13.2).

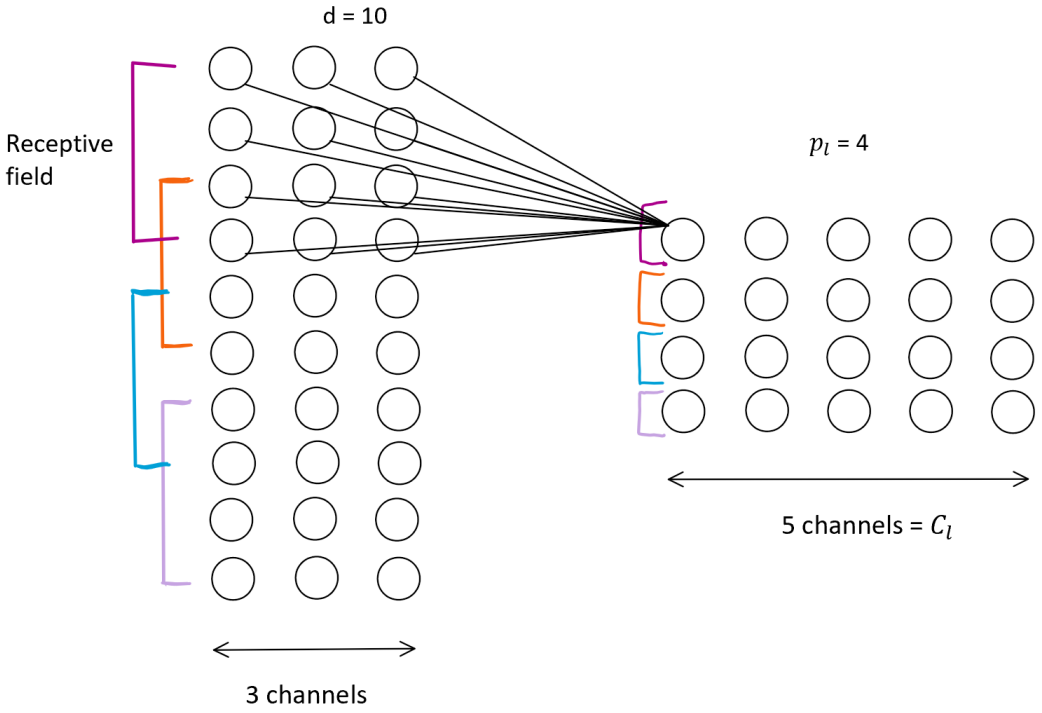


Figure 33: Illustration of the convolutional layer where the input to the layer is one dimensional, with 3 channels, and the output of the layer is 4 dimensional with 5 channels.

13.1.2 Building Blocks of CNN architectures

In this subsection, some more building blocks of the Convolutional Neural Networks are presented involving a set of more detailed graphical explanations.

- **Convolutional layer in 1D:** Figure 34 illustrates an example of what a *convolutional layer* does on a 1 dimensional input from the previous layer. We have one input channel.

  Given an input to the layer of dimension W, we define a filter of size w, usually chosen small. The elements of the filter are the free parameters that will be updated in the optimization process.

  We start by applying the filter on the left side of the input to the layer: The scalar product between the filter weights and the corresponding receptive field (in grey) is computed and the result is stored in the first component of the output of the layer. Then the position of the filter is shifted by a certain value, known as *stride(s)*[25], on the input to the layer (in this particular case the stride is equal to 1) and the same procedure is performed again and again in order to fill all the components in the output of the layer. In the end, the output vector will have dimension $\frac{W-w}{s} + 1$.

  An important thing to notice is that the filter elements always remain the same as long as we shift the filter along the whole input vector, so the values of the weights do not depend on which receptive filter we are looking at. This property is known as *weight sharing*.

- **Convolutional layers 2D:** The situation can now be generalized in higher dimension and when considering multiple channels, as reported in Figure 35. Given W the width and H the height of the input to the layer, we define again the filter with a certain dimension (w, h).

  Now the scalar product between the weights in the filter and the post-activation values in the corresponding receptive field is performed. The result is stored in the output of the layer. The position of the filter is then shifted along the dimensions of the input data according to the stride; different strides may be chosen to perform the shift in the two dimensions. This computation allows obtaining the first channel of the output of the layer; if multiple output channels are considered, the same procedure is performed considering a different filter, where the weights are modified for each of the channels.

[25] As the reader may already have thought, the choice of the stride is again an hyperparameter which can be tuned.

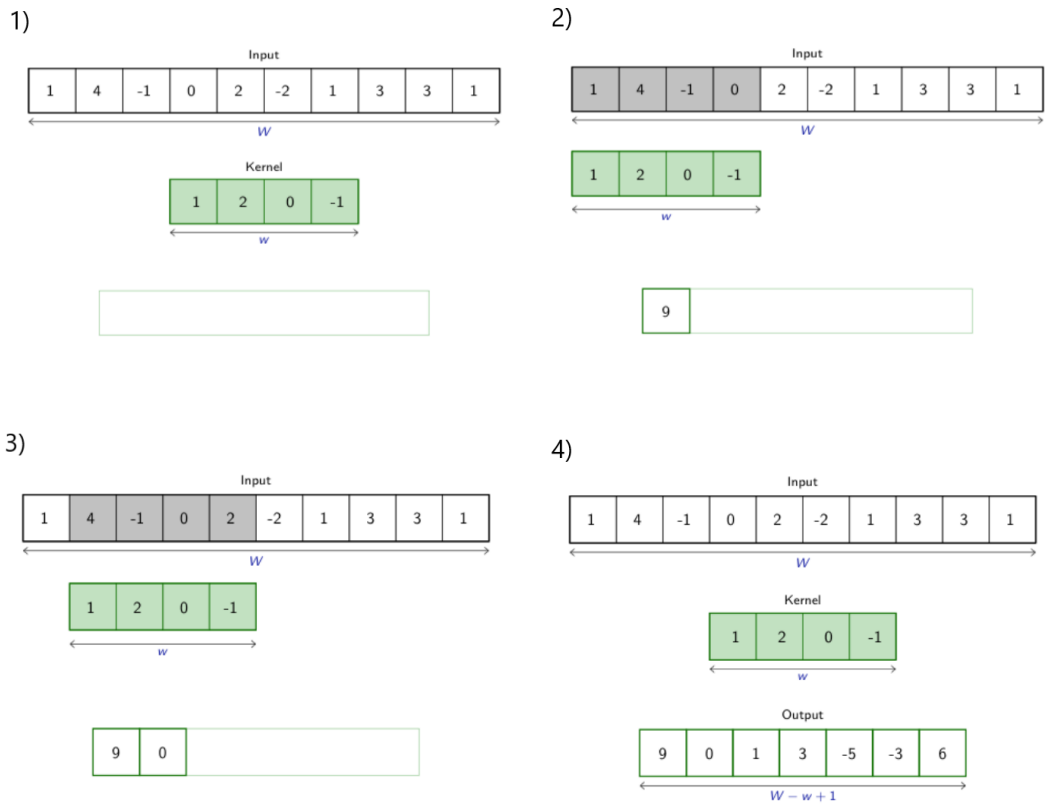


Figure 34: Convolution on 1D input.

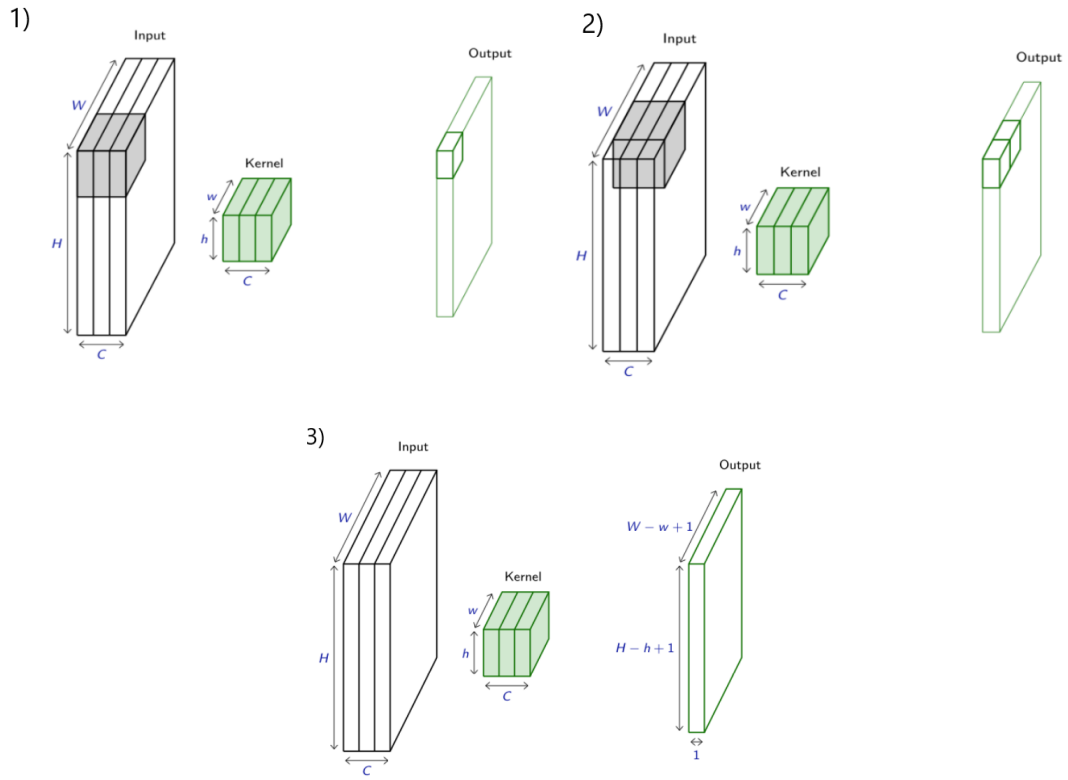


Figure 35: Convolution on 2D input with 3 channels

- **Max Pooling Layers in 1D:** In Figure 36 it is represented how *max-pooling* acts on a 1D vector of length rw. The maximum value among sequential groups of dimension r in the input to the layer is selected and stored in the output of the layer. One can also define the *average-pooling*, where the mean value in the patch is selected and not the maximum.

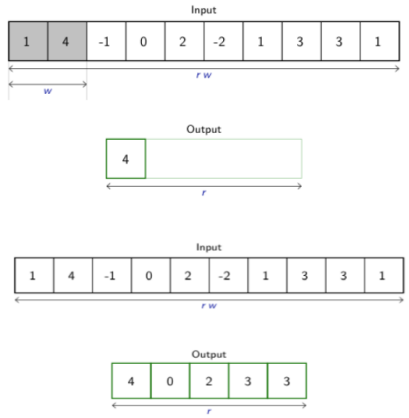


Figure 36: Max-Pooling in 1D data

- **Max Pooling Layers in 2D:** Again the situation can be generalized for higher dimensional inputs to the layer, as shown in Figure 37. The max-pooling is performed for each of the channels of the input separately and in the end the number of channels in the output of the layer will be conserved; this is different from the case of the convolutional layer, where the number of channels typically changes.

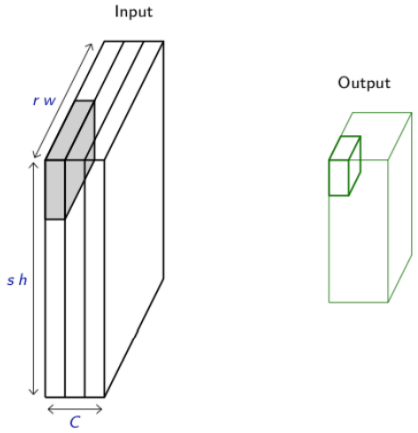


Figure 37: Max-Pooling in 2D data

- **Flatten:** In order to create a fully connected layer, it is necessary to transform the two dimensional image of width w and height h into an one-dimensional vector of length given by the product wh. This can be taken as an input for the Fully Connected Layer.

Further we describe some of the methods that are used to train the network in the context of Convolutional Neural Networks: the *Dropout* and the *Padding*.

- **Dropout:** Some of the neurons in the layers are randomly deactivated, or masked, and their outputs are put to zero, as shown in Figure 38. The mask can be applied to all the input and hidden units in the network and the mask for each unit is sampled independently from all the others. The randomization happens at each

training iteration and a slightly different model with different neuron topology will be created at each iteration, sometimes allowing to avoid overfitting.

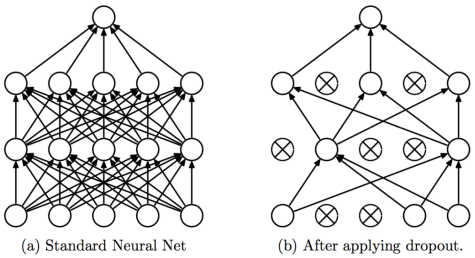


Figure 38: Dropout in a Fully Connected Neural Network: some of the neurons in each layer are randomly deactivated, so their output is put to zero and the connection with the following layer is broken.

- **Padding:** The aim of this feature is to take care of the boundary effects when building convolutional layers. Some pixels with zero values are added in any direction at the border of the image. In this way it is possible to start the convolution in a different position than the corner of the image, allowing to increase the width of the next layer.

13.1.3 Examples of CNN architectures: LeNet and AlexNet

Now, we are going to describe two examples of Convolutional Neural Networks: *LeNet* and *AlexNet*.

- **LeNet:** In figure 39 an example of CNN that was implemented in the 90s and that was interesting for *MNIST digit recognition* is shown. The MNIST dataset is composed of handwritten digits represented in images of 28×28 pixels; the images are black-and-white and so in this case we only have to handle one channel. The database is made of 60'000 training and 10'000 testing images with relative labels from 0 to 9. In this case we have 10 classes to recognize, and the one-hot encoding notation is used to store the information linked to the class an image belongs to.

  The input of the Neural Network will be a $28 \times 28 \times 1$ matrix. Then a convolutional layer with a kernel of dimension 5×5 and padding of size 2 is used to create 6 channels output layer, thus 6 different filters are used. At this layer a non-linear sigmoid function is applied. Then, average-pooling is applied and the dimension of the matrix in each channel is halved.

  These steps are repeated a second time, considering different dimensions at each step, before flattening the correspondent matrix and going to dense fully connected layers. Three of these are applied, where the sigmoid is the correspondent activation function; in the end the last layer has a width equal to the number of classes we want to recognize, in this case 10 as we are performing a classification task. In this final layer, the neuron with the highest value will be the one corresponding to the predicted class.

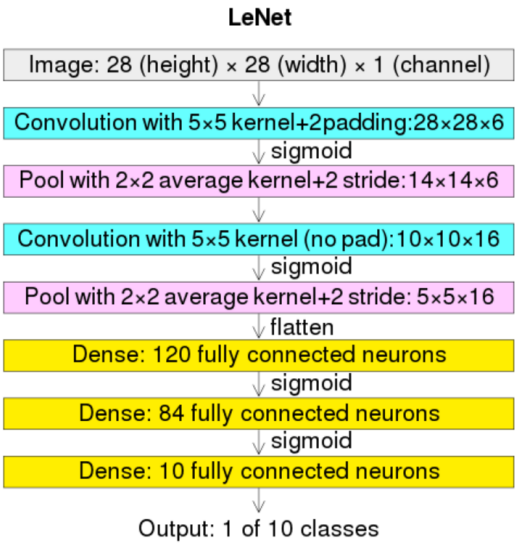


Figure 39: LeNet CNN architecture.

- **AlexNet:** The model in Figure 40 has been proposed in 2012 and it was a revolution in the field. The dataset ImageNet was used in this case, made of higher resolution, and coloured images (higher number of pixels and three channels) which could represent different objects.

  Starting from an input of dimension $(224 \times 224 \times 3)$, 5 convolutional layers are applied, interspaced with pooling, padding and ReLU activation functions. Then, an image with 5×5 pixels and 256 channels are flattened and passed to three fully connected layers, whose final width is in this case 1000. In the dense fully connected layers, the ReLU function is again used as activation and a dropout with $p = 0.5$ is applied, meaning that half of the outputs in the layers will be masked. The last layer has 1000 neurons, from which we can retrieve the output and the predicted label.

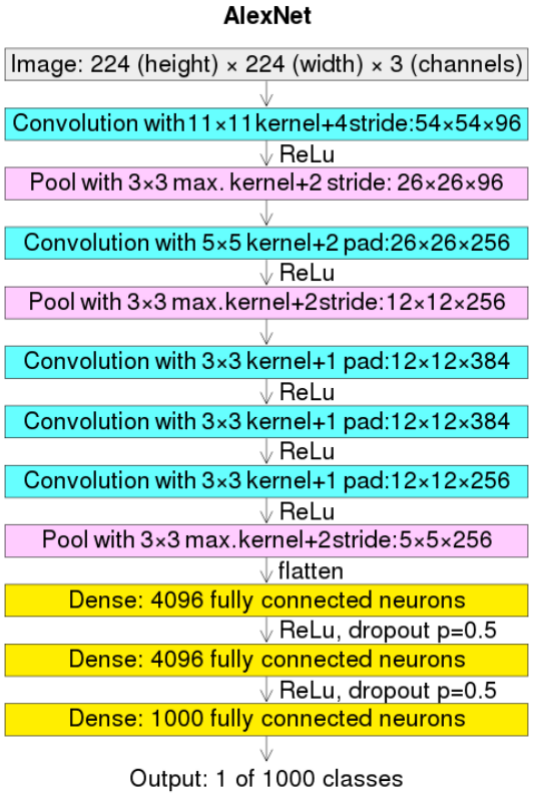


Figure 40: AlexNet CNN architecture

In Figure 41 we show the performance in classification of *ImageNet* experiments over the years, using different models and increasing the number of layers. Starting form a percentage error bigger than 20% in 2011 where shallow neural networks where used, with *AlexNet* an dramatic improvement of almost 10% occurred in 2012, using 8 layers. Then, by using again a similar architecture for convolutional neural networks and increasing the number of layers, in the following years other improvements occurred, leading to an error percentage around 7% in 2014. An interesting example is represented by *VGG*; in all these cases up to 22 layer were used. Then, another important advance occurred in 2015 with *ResNet*, leading to a percentage error of 3.57%. In this case the number of layers used is 152.

Generally, CNNs are faster to train and are more computationally friendly than fully connected neural networks as some weights are frozen and many of them are the same (weight sharing). Despite of this, when increasing the number of layers and the dimension of the networks, the computational cost may be still high. In the case of ResNets, a trick was used to address this problem: When connecting neurons some layers are skipped. This means that a given neuron is not connected with the one of following layer but with one neuron of some layers later. This allowed to use a bigger number of layers and to obtain a significant improvement in the classification of images.

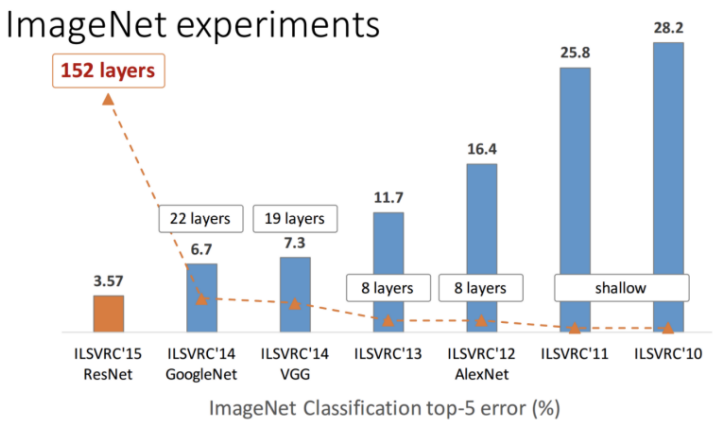


Figure 41: Classification top-5 error in percentage for the ImageNet experiment

LZ: Add notes on transformer layers.

13.2 Deep Learning: modus operandi

Let us discuss some of the properties of current neural networks that appear across architectures and datasets.

13.2.1 Overparameterized Networks and Interpolation

Recall the classical bias-variance trade-off principle:

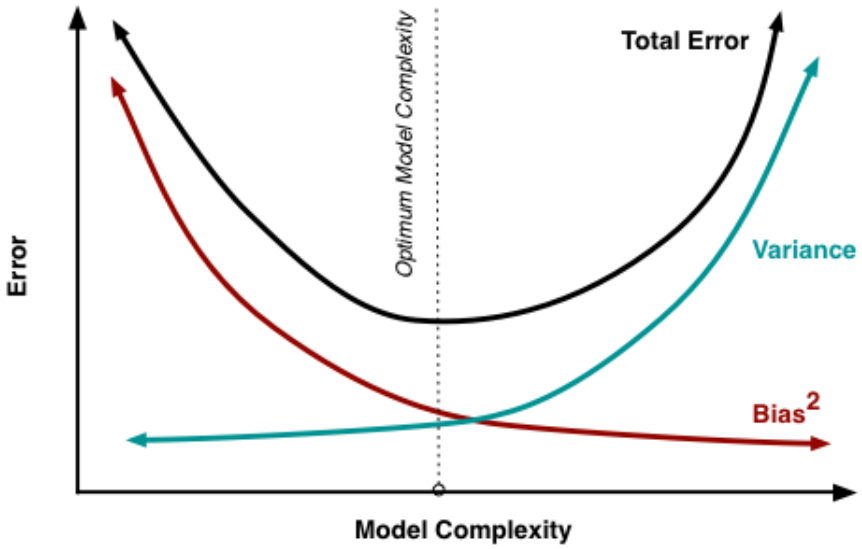


Figure 42: Classical bias-variance trade-off

The blue curve is the variance, or "how much we fit the noise". The red one is the bias, of "how far is the model class from the truth".

So, Figure 42 represents the test error as a function of the model size (or complexity), which corresponds to the number of parameters (trained weights) of the model. We obtain a "U-shaped" curve, with the optimum model complexity corresponding to the curve minimum, also called the *sweet spot*: At this point the model is not too complex which would lead to overfitting and nor too simple which would lead to underfitting.

Consider now neural networks trained with no regularization. The new error curve is shown in Figure 43.

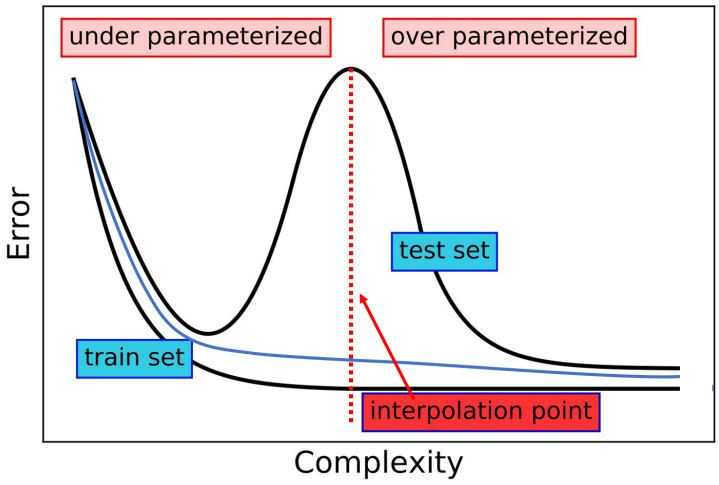


Figure 43: Bias-variance trade-off for trained neural networks: The *double-descent curve*. The upper black curve corresponds to the *test error*, the lower black curve to the *train error* that becomes zero at the *interpolation point*. The blue curve is the test error of regularized or early stopped training.

The upper black curve is the test error, while the lower black one is the train error. The graph is clearly separated in two parts by the vertical dotted line: the left area, where that training error remains positive, corresponds to the previous figure, the model is underparameterized. One can see again the U-shaped test error.

The right area however corresponds to an overparameterized regime, the training error is 0: indeed, all training samples are perfectly fit to their labels, so we are *interpolating* the training set. This region is defined such that the training error is zero. The number of parameters is in general larger than the number of samples n in this region, implying a high model complexity.

The separation point between those two areas is called the *interpolation point*. In learning with random features, discussed in lecture 11, the interpolation happens when the size of the hidden layer p exceeds the number of samples n. However, one should note that in general the interpolation point does not coincide with the point where the number of parameters is equal to the size of the training set. The test error at this point is called the *interpolation peak*. The blue curve is a typical test error but with regularization or early stopping, which reduces the peak. This type of graph for neural networks is called a *double-descent curve*.

13.2.2 State-Of-The-Art (SOTA) Neural Networks and Implicit Regularisation

Current State-Of-The-Art (SOTA) neural networks usually have several parameters greater than the number of samples over which they are trained. They operate in the overparametrized regime visible on Figure 43. In the latter, all training samples are consistently fitted. The only effect of adding parameters is to enlarge the space of solutions, leaving one to wonder about the gains of doing so. Many papers have shown that overparametrized neural networks work really well. In fact, it somehow happens that the combinations of architectures and training algorithms lead to a good test error, with no overfitting nor explosion of the variance. This idea that the algorithm implicitly chooses a path leading to a good area in the space of weights defines the so-called *implicit regularisation*. And it is precisely by adding parameters to a model that its *implicit bias improves*. The existence of these phenomena explains why, when defining the loss of neural networks' in section 11.3 and Eq.(11.8), no regularisation is added (since it already happens spontaneously). It must be noted that there is still a lot of ongoing research to understand both the *double-descent behavior* and *implicit regularisation*.

The following Figure 44 and Figure 45 showcase the 'double-descent behavior' of a neural network. The last panel of Figure 44 in particular, constitutes an example of neural network that works in the overparametrized (interpolation) regime.

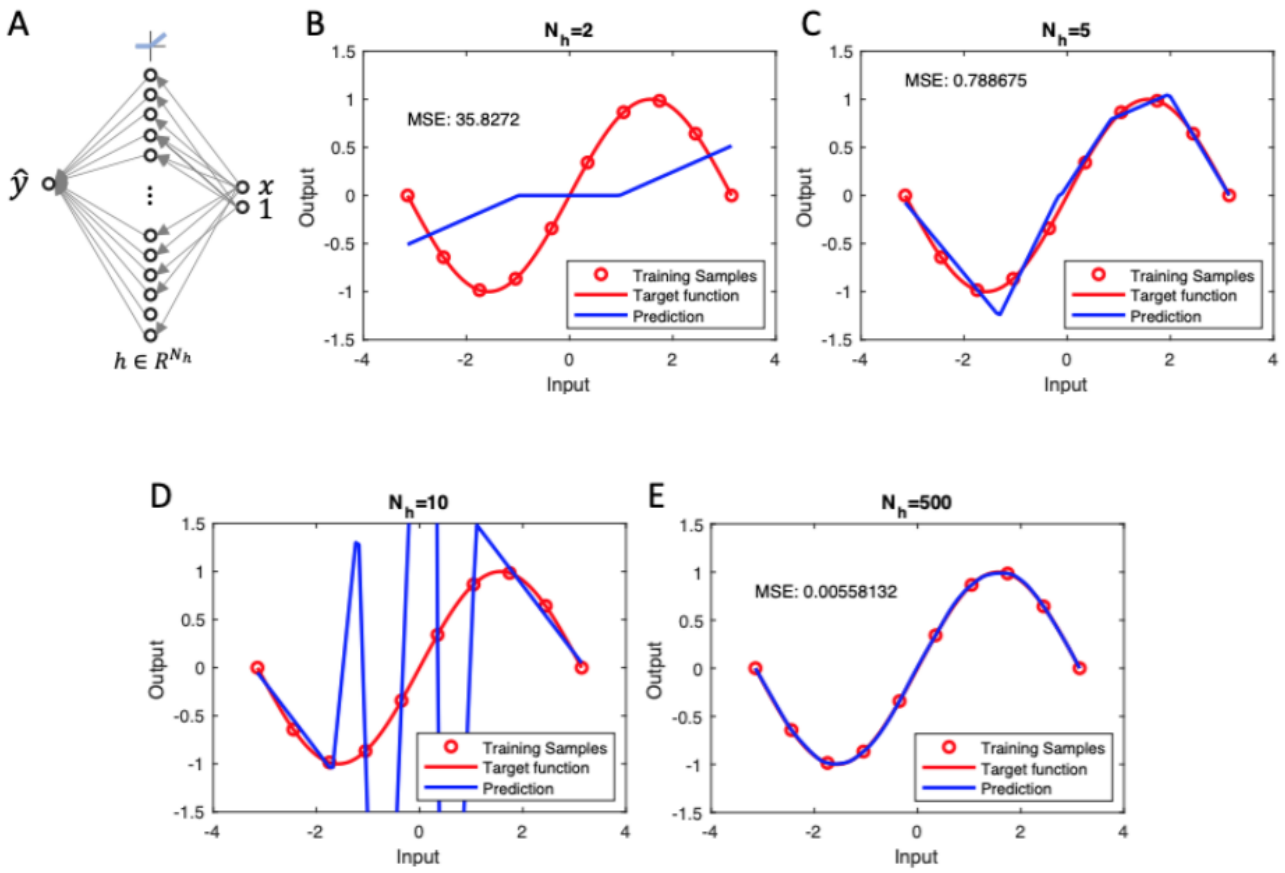


Figure 44: overparameterization and generalization (from a talk by Andrew Saxe). Sub-figure **(A)** illustrates a simple ReLU network with random first layer weights and trained second layer weights. The network receives a scalar input and bias term and is trained to minimise squared error on ten points (red circles) from a target sinusoid (red curve) (see panels **(B-E)**). In sub-figures **(B)** to **(E)**, the blue curves show example functions learned by networks with different numbers of hidden units $\mathbf{N_h} \in \{2, 5, 10, 500\}$. Networks in panel **(B)**, **(C)** and **(D)** show the standard progression from underfitting the training data to overfitting it, with a happy medium in panel **(C)**. They showcase the classical bias-variance trade-off behaviour (see Figure 42) that characterizes the underparameterized regime of the double-descent curve (see Fig. 43). However, the large network panel **(E)** generalizes the best. This network has 50 times more parameters than training examples but generalizes well, because among the infinity of solutions attaining zero training error a low norm solution has been chosen. It illustrates the overparametrized regime of the double-descent curve (see Figure 43).

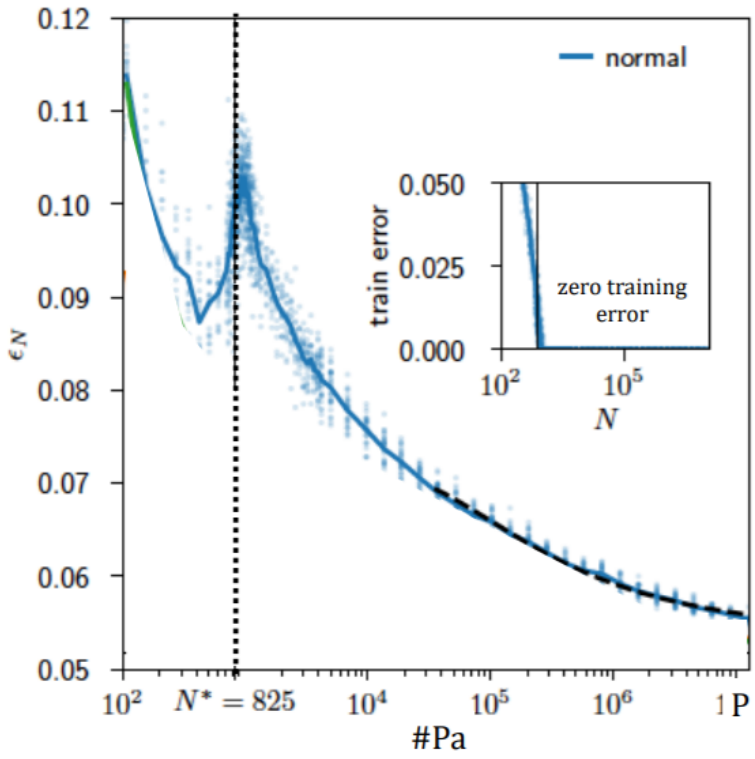


Figure 45: Test error ϵ_N for the MNIST-Parity dataset obtained in function of #Pa, the number of parameters in a 5 layer fully connected network and for a hinge loss with no regularisation. Figure taken from [17].

Remark that not all arbitrary overparametrized networks yield a good test error. They might fail. But those that engineers have found to have a good performance also have a huge overparameterization (without overfitting). In particular, in Figure 46 the bigger the network and the more overparametrized it is, the lower the test error. So overall, neural networks are overparametrized and despite this they work well in many cases.

Furthermore, neural networks have so many parameters that even when selecting labels at random, zero training error is still obtained (see Figure 47). This discovery made in 2016 by Zhang, Bengio, Hardt, *et al.*, was both revolutionary and devastating for theoreticians in the field as most of the theory in computer science was based on the assumption that commonly used learning methods were not able to fit random labels to zero train error [18]. So this discovery not only rendered void most of existing 'classical' theories explaining the functioning of neural networks, but it has also highlighted the mathematical challenges of understanding how neural networks work and the necessity to develop a theory in this field.

13.2.3 Bad global minima exist

To this day it is still a mystery how we manage to train neural networks using only SGD. In fact, it was and is unexpected that SGD works so well in a loss landscape that is intrinsically non-convex (because the activations are non-linear and the weights to find are present in all possible layers). Recall that this behavior of algorithms converging to a minimum with a good test error is called implicit regularization. To explain the latter, one could naively hypothesise that, depending on the way the algorithm is constructed, there is only one minimum of the training loss that generalizes well (so that using GD to find it is fairly easy). But when investigating such conjecture, we arrive at the conclusion that it is not true. In fact, Achlioptas et al. (2019) have invented a way to construct global minimum with zero training loss and very bad test error, that completely invalidates the above hypothesis [19].

Their 'algorithm' works in the following way: They start by fitting the weights to randomly labeled data. Then they replace the random labels by the true labels little by little, retraining the weights everytime such as to keep the training error to zero. By doing this they manage to attain the so-called *bad global minima*. To explain this, the paper actually shows that, when starting with random labels, the GD lands in a point of weight space where the test error is really bad. Then when retraining the weights with the correct labels, the algorithm more or less remains in the same region of weight space, explaining why, even when using the correct training labels, the test remains pretty bad.

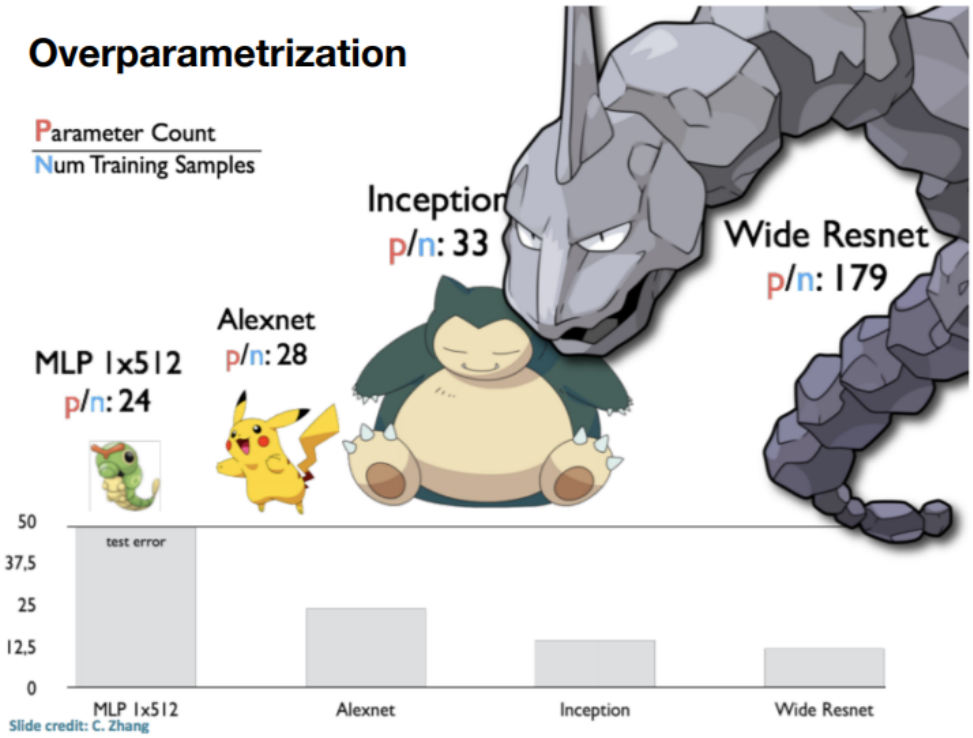


Figure 46: Test error in function of different classification algorithms with different ratios between the number of parameter and training samples.

Actually, this is not surprising. In fact, when a neural network is overparametrized, there are so many ways to set the weights such that they lead to zero training error that it is expected that some minima give good test errors and others really bad ones. Nonetheless, the fact that when starting the algorithm randomly with the correct labels, the point to which the GD converges is completely different than when running the above-described algorithm. This proves once again that there is something non-trivial happening in the trajectories of algorithms. Sometimes, the trajectories converge to good minima (implicit regularization) and other times to bad ones. Anyhow, trajectories of overparametrized neural network algorithms remain to be investigated and researched about.

LZ: Add notes on transfer learning.

13.2.4 Adversarial Examples

An adversarial example is an instance with small, intentional feature perturbations that is applied to a specific input to cause a machine learning model to make false predictions. The perturbation that is applied to the input image, which may be imperceptible to humans, is created in a correlated way such that the correctly trained neural network gives a specific wrong result. Moreover, it is not necessary for the perturbation to be applied to the whole image: Even a partial modification of it may lead to a wrong classification. An example of this is reported in Figure 48. Whether it is possible to build neural networks that are robust to this kind of attacks is still an open question and much research is carried out on this topic as this brings into question the readiness of neural networks for security-critical applications.

13.2.5 Data Augmentation

What can we do when there is not enough data? Simply create new ones. This is the principle of data augmentation. Firstly, pixels are changed but not the label, then the training is done on the transformed data. The transformation of the data can be for example to flip the image, to randomly crop, to add contrast, color jitters or brightness randomly as we can see on the Figure 49. All the transformation techniques can be mixed. Domain knowledge helps in finding new data augmentation techniques. This technique is very useful for small datasets.

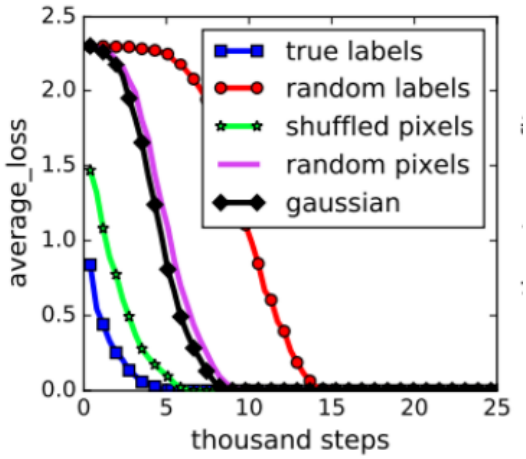


Figure 47: Fitting random labels and random pixels on CIFAR10. This figure shows the training loss of various experiments settings decaying with the training steps [18].

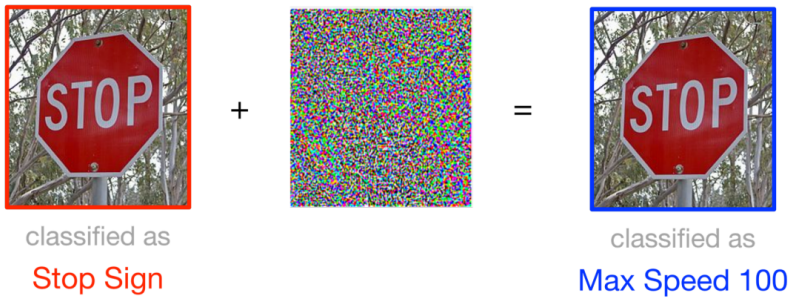


Figure 48: Adversarial example over a stop sign. In this particular case, even a small picture over the sign may lead to a wrong classification from the recognition software of the car, leading to unintended consequences such as a car crash.

Figure 49: Example of transformation for data augmentation

14 Lecture 14: Unsupervised learning using deep learning tools 24.12.

(pre-recorded) Students designated to write this section: Balsiger David Joachim, Azzoug Nour, Astruc Lopez Alejandro, Arbez Hugo Paul, Aragon Antoine Louis Marie, Abdallah Mohamed Samy Mohamed Tawfik
Coordinator:
TA: Alessandro (TA approved)
Students who actually contributed: Azzoug Nour, Astruc Lopez Alejandro, Arbez Hugo Paul, Aragon Antoine Louis Marie, Abdallah Mohamed Samy Mohamed Tawfik

In this last lecture, we consider ways to leverage some tools and tricks of supervised deep learning that we have covered in the previous few lectures for unsupervised learning.

14.1 Recapitulative on Boltzmann Machine

This section is a reminder of the Boltzmann machine, which we have covered in Lecture 9. We consider a number n of d-dimensional binary samples $X_\mu \in \{\pm 1\}^d$, with $\mu = 1, \dots, n$. The goal of the Boltzmann machine is to generate sequences of data $x_{\text{new}} \in \{\pm 1\}^d$ that look like those in the training set, meaning that x_{new} should be distributed similarly to $\{X_\mu\}_{\mu=1}^n$.

The Boltzmann machine assumes the Boltzmann probability distribution for x, which is given by:

$$p_{h,J}(x) = \frac{1}{Z} e^{\sum\limits_{i<j} J_{ij} x_i x_j + \sum\limits_{i=1}^{d} h_i x_i} \tag{14.1}$$

where Z is the partition function, and the values of J_{ij} and h_i are found through training the model on the given data points X. As it was presented in Lecture 9, training is practically done imposing the following constraints:

$$\begin{aligned} \langle X_i \rangle_{\text{emp}} &= \tfrac{1}{n} \sum_{\mu=1}^{n} X_{\mu i} = \langle x_i \rangle_{p_{h,J}(x)} \\ \langle X_i X_j \rangle_{\text{emp}} &= \tfrac{1}{n} \sum_{\mu=1}^{n} X_{\mu i} X_{\mu j} = \langle x_i x_j \rangle_{p_{h,J}(x)} \end{aligned} \tag{14.2}$$

where the first equation is for the empirical average of X_i, it leads to learning h_i and the second equation is for the two point correlation of $X_i X_j$, it leads to learning J_{ij}. As shown in Eq.(9.21):

$$h = -\Sigma^{-1}\mu, \;\; J = -\Sigma^{-1}\,.$$

This choice of constraints can be justified with the maximum entropy principle. By applying gradient descent on the likelihood, we obtained the equations:

$$\begin{aligned} h_i^{t+1} &= h_i^t - \gamma[\langle X_i \rangle_{\text{emp}} - \langle x_i \rangle_{p_{h,J}(x)}] \\ J_{ij}^{t+1} &= J_{ij}^t - \gamma[\langle X_i X_j \rangle_{\text{emp}} - \langle x_i x_j \rangle_{p_{h,J}(x)}] \end{aligned} \tag{14.3}$$

where the averages over the probability distribution $p(x)$ are computed by running a Monte Carlo Markov Chain (MCMC) simulation. This concludes what we have already covered in Lecture 9. In order to make the Boltzmann machine more expressive, we consider adding hidden variables to the model.

14.2 Adding hidden variables to the Boltzmann Machine

Let's define *visible* and *hidden variables* respectively, as:

$$x_i \in \{\pm 1\}, \quad i = 1, ..., d \tag{14.4}$$

$$u_i \in \{\pm 1\}, \quad i = 1, ..., p. \tag{14.5}$$

We can now postulate the following probability distribution:

$$P_\theta(x) = \frac{1}{Z} \sum_{\{u_i\}_{i=1}^p} e^{E_\theta(\{x_i\}_{i=1}^d, \{u_i\}_{i=1}^p)}, \tag{14.6}$$

where

$$E_\theta(x,u) = \sum_{i<j} J_{ij} x_i x_j + \sum_{i=1}^{d} h_i x_i + \sum_{i,j} \tilde{J}_{ij} x_i u_j + \sum_{i<j} \tilde{\tilde{J}}_{ij} u_i u_j + \sum_{i=i}^{p} \tilde{h}_i u_i, \tag{14.7}$$

and $\theta = \{J_{ij}, \tilde{J}_{ij}, \tilde{\tilde{J}}_{ij}, h_i, \tilde{h}_i\}$. Since equation (14.6) depends on more parameters than the regular Boltzmann machine, it is more general and can represent more complex probability distributions.

To train the Boltzmann machine with hidden variables, we follow the same steps as before. We begin by writing the maximum likelihood:

$$\begin{aligned}\mathscr{L}(\theta) &= \prod_{\mu=1}^{n} P_\theta(X_\mu) \\ &= \prod_{\mu=1}^{n} \left[\frac{1}{Z(\theta)} \sum_u e^{E_\theta(X_\mu, u)} \right] \\ \frac{1}{n} \log \mathscr{L}(\theta) &= \frac{1}{n} \sum_{\mu=1}^{n} \log \left(\sum_u e^{E_\theta(X_\mu, u)} \right) - \log Z(\theta).\end{aligned} \tag{14.8}$$

Taking the gradient with respect to a single parameter θ, e.g., θ could be J_{ij} or $\tilde{h}_i$, we obtain:

$$\begin{aligned}\frac{1}{n} \frac{\partial \log \mathscr{L}(\theta)}{\partial \theta} &= \frac{1}{n} \sum_{\mu=1}^{n} \frac{\sum_u \frac{\partial E_\theta}{\partial \theta} e^{E_\theta(X_\mu, u)}}{\sum_u e^{E_\theta(X_\mu, u)}} - \frac{\partial}{\partial \theta} \log Z_\theta \\ &= \underbrace{\frac{1}{n} \sum_{\mu=1}^{n} \left\langle \frac{\partial E_\theta(X_\mu, u)}{\partial \theta} \right\rangle_{P_\theta(X = X_\mu, u)}}_{\text{fix } X_\mu, \text{ sample } u} - \underbrace{\left\langle \frac{\partial E_\theta(x, u)}{\partial \theta} \right\rangle_{P_\theta(x,u)}}_{\text{sample both } x \text{ and } u}.\end{aligned} \tag{14.9}$$

The average where we fix X_μ is solely with respect to the hidden variables u_i, and is often referred to as the *clamped average*. The update equation for θ becomes:

$$\theta^{t+1} = \theta^t + \gamma \left[\left\langle \frac{\partial E_\theta}{\partial \theta} \right\rangle_{\text{clamped}} - \left\langle \frac{\partial E_\theta}{\partial \theta} \right\rangle_{P_\theta(x,u)} \right]. \tag{14.10}$$

In practice, however, training these models is quite cumbersome, and therefore they are not very used in practice.

14.2.1 Restricted Boltzmann Machine (RBM)

The most widely used version of the Boltzmann Machine with hidden units is the *Restricted Boltzmann Machine* (RBM). In this special case, the energy takes into account only interactions between the visible and hidden units:

$$E_\theta(x, u) = \sum_{ij} J_{ij} x_i u_j + \sum_{i=1}^{d} h_i x_i + \sum_{i=1}^{p} \tilde{h}_i u_i. \tag{14.11}$$

In particular, there are no visible-visible nor hidden-hidden interactions.

The clamped average for the RBM can be written explicitly in closed form (there is no need of MCMC). For instance, if $\theta = J_{ij}$

$$\langle X_{\mu j} u_i \rangle_{\text{clamped}} = X_{\mu j} \tanh \left(\sum_{l=1}^{n} X_{\mu l} J_{li} + h_i \right) \tag{14.12}$$

which corresponds to a model of non-interacting spins. The clamped average is often called *positive gradient*. The model average $\langle x_i u_j \rangle$, or *negative gradient*, can be efficiently computed by using MCMC sampling:

1. u is sampled with x fixed.
2. x is sampled with u fixed.
3. Repeat.

All in all,

$$J_{ij}^{t+1} = J_{ij}^t + \gamma[\text{Positive Gradient} - \text{Negative Gradient}]. \tag{14.13}$$

This model works nicely and is very used in practice.

14.2.2 Deep RBM

We can also add layers of hidden units to the RBM. In the three-layers case, the energy reads

$$E_\theta(x, u^{(1)}, u^{(2)}) = \sum_{ij} J_{ij}^{(1)} x_i u_i^{(1)} + \sum_{ij} J_{ij}^{(2)} u_i^{(1)} u_j^{(2)} + \sum_{i}^{d} h_i x_i + \sum_i h_i^{(1)} u_i^{(1)} + \sum_i h_i^{(2)} u_i^{(2)} \tag{14.14}$$

where the interactions are between the first and second layer, and the second and third layer. However, these deeper models see limited use nowadays.

14.3 Auto-Encoder

Now, we look at two different ways to leverage f*eed-forward neural networks* for *unsupervised learning* and, in particular, *data generation*. Firstly, we will study *auto-encoders* and, secondly, *Generative Adversarial Networks* (GANs).

14.3.1 Architecture of Auto-Encoders

Auto-encoders are a way to generate samples $\tilde{X}$ which have close distributional properties to the original data X, meaning that it should be difficult to discriminate the original data from the fake artificially generated data. Additionally, auto-encoders can be used to learn a low-dimensional representation of the data. Their principle is to use the data itself as labels and then train a multi-layer neural network on this new dataset $\{X_\mu, y_\mu = X_\mu\}_{\mu=1}^n$. This multi-layer neural network is divided into two parts: the *encoder* and the *decoder*. The encoder takes the data $X_\mu \in \mathbb{R}^d$ as input and outputs a compressed representation $Z_\mu \in \mathbb{R}^p$, with $p \ll d$. The decoder takes Z_μ as input and outputs $\tilde{X}_\mu \in \mathbb{R}^d$. An example of an auto-encoder can be observed on Figure 50.

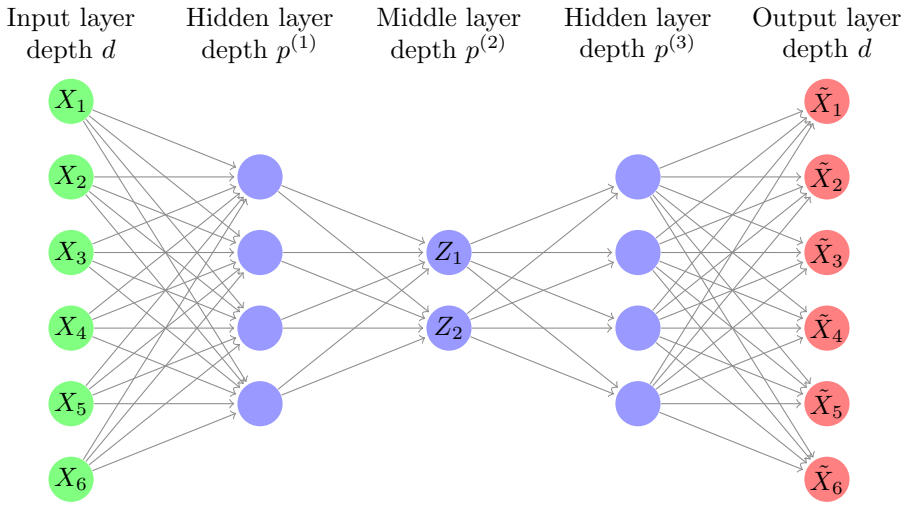


Figure 50: Representation of an *auto-encoder* for which the input data has dimension $d = 6$ and the middle layer has $p = p^{(2)} = 2$. The layers from the input layer up to the middle layer (included) form the encoder, and the decoder consists of all subsequent layers.

Auto-encoders can be trained by using the *mean squared loss*:

$$\mathscr{L}(W) = \sum_{\mu=1}^{n} l(X_\mu; f_W(X_\mu)) = \sum_{\mu=1}^{n} \sum_{i=1}^{d} (X_{\mu i} - [f_W(X_\mu)]_i)^2 \tag{14.15}$$

where the input data X_μ play the role of the labels and $f_W(X_\mu)$ is the output function of a multi-layer neural network, as in Equation (11.15).

14.3.2 Applications of Auto-Encoders

A first example of the use of an auto-encoder is dimensionality reduction. We train the network on the data by minimizing the loss with gradient descent. Then a reduced version of the data can be obtained by cutting out the

trained decoder part and feeding the initial data X to the trained encoder part only. The output Z is a reduced representation of the data. This idea can be used to pre-process the data, or as a pre-training by re-using the encoder part of the network in a larger network.

A second example is the use of the auto-encoder as a generative model as the RBM. This is based on the fact that we can choose p, the middle layer dimension, such that we have an optimal compression, and the middle layer has i.i.d. post activations, since all correlations should have been removed. Then, when the full auto-encoder network has been trained, the encoder part is cut out, and we can replace Z_i by random i.i.d. variables. Feeding such i.i.d. variables to the decoder part outputs data X_{new} that should look like the original data.

Moreover, auto-encoders can be used to remove the noise in an image or a signal. Starting from the original data, the idea is to corrupt the inputs with noise and instead of using the corrupted inputs as labels in the loss, to use the original data. Therefore, the auto-encoder will learn to reconstruct the original data, and after training it can be used to denoise new samples. An example of the application of the results obtained by using a denoising auto-encoder can be observed in Figure 51.

Figure 51: Effect of a denoising auto-encoder on old pictures.

14.4 GANs

The idea behind *Generative Adversarial Networks* (GANs) is to train a neural network, called the generator, to generate artificial data in such a way that another neural network, called the discriminator, has a hard time to distinguish true data from the data generated by the generator.

G : Generator neural network This neural network f_G with weights W_G takes a random input $Z \in \mathbb{R}^d$, and generates the output $\widetilde{X}$:

$$\widetilde{X} = f_G(W_G, Z)\,. \tag{14.16}$$

D : Discriminator neural network This neural network f_D with weights W_D takes inputs that can be either a real sample X_μ or a sample $\widetilde{X}_\mu$ generated by the generator networks. Labels are assigned depending on the origin of the samples, i.e., $y_\mu = +1$ for the true data and $y_\mu = -1$ for the generated data. The discriminator returns the output:

$$f_D(W_D, (X, \widetilde{X})) \tag{14.17}$$

with $(X, \widetilde{X})$ the concatenated real and generated data.

Training a GAN In order to train a GAN, we introduce the loss function :

$$\mathscr{L}(W_D, W_G) = \frac{1}{2n}\sum_{\mu=1}^{2n} l(y_\mu;\ f_D(W_D, (X, \widetilde{X}))) = \frac{1}{2n}\sum_{\mu=1}^{2n} l(y_\mu;\ f_D(W_D;\ (X, f_G(W_G, z_\mu))))\,. \tag{14.18}$$

In order to optimize the performance of the discriminator, we minimize the loss function with respect to its weights W_D. Moreover, since the generator is trying to make the task harder for the discriminator, we maximize the loss function with respect to W_G, obtaining the following *max-min problem* formulation:

$$\max_{W_G}\min_{W_D} \mathscr{L}(W_D, W_G). \tag{14.19}$$

The details of this optimization procedure can be daunting, and often there are convergence problems. Nevertheless, there are many successful examples of the use of GANs. For instance, these models can generate impressive artificial pictures of people faces; with only a visual inspection, we are not able to tell that they were generated by an algorithm. Figures 52 and 53 represents two non-existing people artificially created by a GAN.

Figure 52: Picture generated by GANs - Example 1

Figure 53: Picture generated by GANs - Example 2

https://www.overleaf.com/project/614466b6274a8d1bd6eb3d74

References

[1] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[2] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep Blue," en, *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, Jan. 2002. (visited on 10/01/2021).

[3] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski, "Temporal difference learning of position evaluation in the game of go," *Advances in Neural Information Processing Systems*, pp. 817–817, 1994.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," en, *Nature*, vol. 550, no. 7676, Oct. 2017. (visited on 10/01/2021).

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018, Publisher: American Association for the Advancement of Science. (visited on 10/01/2021).

[6] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering Atari, Go, chess and shogi by planning with a learned model," en, *Nature*, vol. 588, no. 7839, Dec. 2020, ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. [Online]. Available: https://www.nature.com/articles/s41586-020-03051-4 (visited on 10/01/2021).

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," en, *Nature*, vol. 529, no. 7587, Jan. 2016, ISSN: 1476-4687. DOI: 10.1038/nature16961. [Online]. Available: https://www.nature.com/articles/nature16961 (visited on 10/01/2021).

[8] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. R. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P. Kohli, D. T. Jones, D. Silver, K. Kavukcuoglu, and D. Hassabis, "Protein structure prediction using multiple deep neural networks in the 13th Critical Assessment of Protein Structure Prediction (CASP13)," *Proteins: Structure, Function, and Bioinformatics*, vol. 87, no. 12, pp. 1141–1148, 2019. (visited on 10/01/2021).

[9] ——, "Improved protein structure prediction using potentials from deep learning," *Nature*, vol. 577, no. 7792, pp. 706–710, Jan. 2020. (visited on 10/01/2021).

[10] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with AlphaFold," en, *Nature*, vol. 596, no. 7873, pp. 583–589, Aug. 2021. (visited on 10/01/2021).

[11] *Deepl translate: The world's most accurate translator*. [Online]. Available: https://www.deepl.com/en/translator.

[12] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to machine learning for physicists," *Physics Reports*, vol. 810, pp. 1–124, May 2019.

[13] R. Penrose, "A generalized inverse for matrices," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 51, no. 3, pp. 406–413, 1955.

[14] J. Novembre, T. Johnson, K. Bryc, Z. Kutalik, A. R. Boyko, A. Auton, A. Indap, K. S. King, S. Bergmann, M. R. Nelson, *et al.*, "Genes mirror geography within europe," *Nature*, vol. 456, no. 7218, pp. 98–101, 2008.

[15] P. De Los Rios and P. Goloubinoff, "Chaperoning protein evolution," *Nature Chemical Biology*, vol. 8, 2012.

[16] K.-I. Funahashi, "On the approximate realization of continuous mappings by neural networks," *Neural networks*, vol. 2, no. 3, pp. 183–192, 1989.

[17] M. Geiger, A. Jacot, S. Spigler, F. Gabriel, L. Sagun, S. d'Ascoli, G. Biroli, C. Hongler, and M. Wyart, "Scaling description of generalization with number of parameters in deep learning," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2020, no. 2, p. 023 401, Feb. 2020.

[18] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *CoRR*, vol. abs/1611.03530, 2016. arXiv: 1611.03530. [Online]. Available: http://arxiv.org/abs/1611.03530.

[19] S. Liu, D. S. Papailiopoulos, and D. Achlioptas, "Bad global minima exist and SGD can reach them," *CoRR*, vol. abs/1906.02613, 2019. arXiv: 1906.02613. [Online]. Available: http://arxiv.org/abs/1906.02613.