Deep Learning and Applied Artificial Intelligence

Sapienza University of Rome

A.Y. 2021-2022

Donato Crisostomi

Student lecture notes for the *Deep Learning and Applied Artificial Intelligence* course, held by Prof. E. Rodolà.

Disclaimer and acknowledgments

This document served as my personal course notes when I took the course in A.Y. 19/20; if I will have time, I will try to update it to reflect the more recent material seen throughout the course. Most of the material comes from the course lectures held by Prof. E. Rodolà, with some additions here and there. The document is not guaranteed to be error-free, and should always be taken *cum grano salis*. Please reach out to me to report errors and/or problems.
A special thanks to S. Antonelli, G. Attenni, A. Caciolai and S. Esposito who helped me to draw up these notes.

Contents

Chapter 1

Data

Machine learning involves dealing with *data*. The first thing to do when facing a problem involving data is to look at the data.
In fig. 1.1 each dataset has the same summary statistics, but those are not sufficient to reveal the underlying structure; indeed, given those statistics, one would probably imagine those datasets like in fig. 1.2.

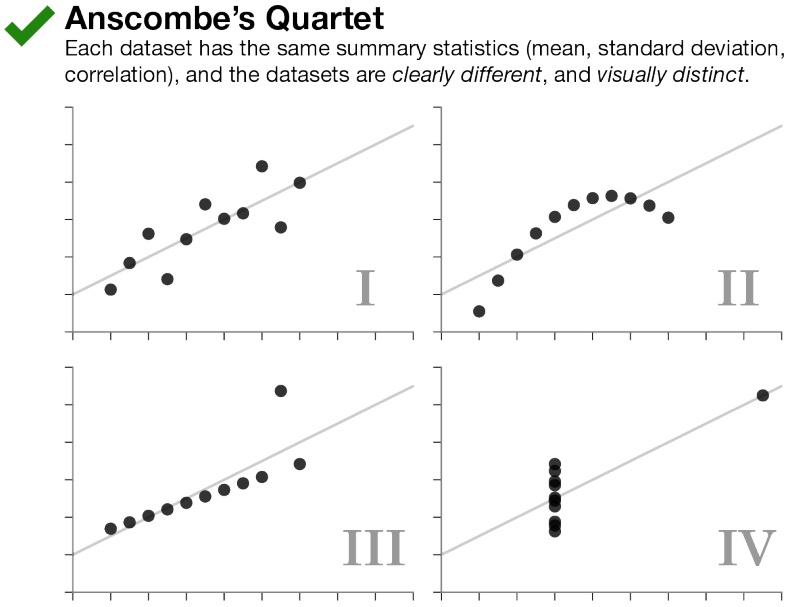


Figure 1.1: Structured quartet.

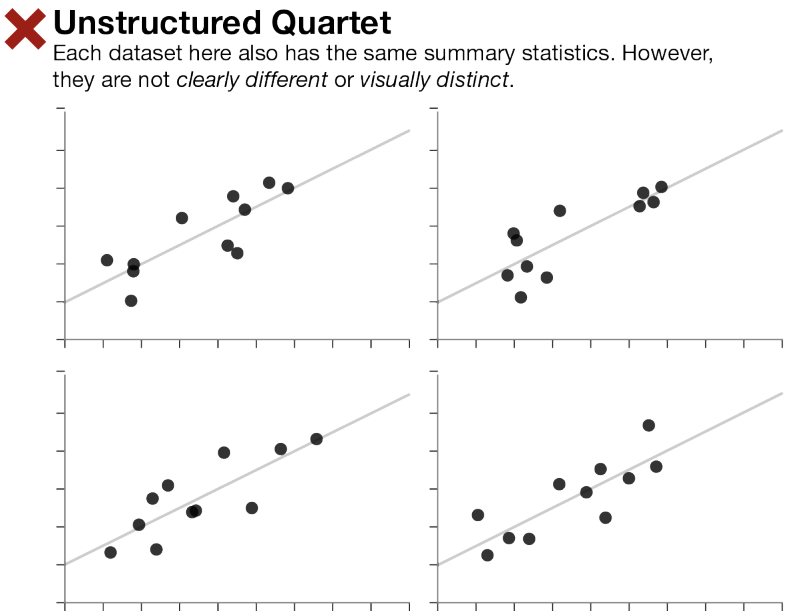


Figure 1.2: Unstructured quartet.

Another example of how summary statistics can be misleading is fig. 1.3, in which all the datasets have the same summary stats to 2 decimal places but totally different structures. Again, working with these datasets without a proper visualization would result in missing the underlying structures. Deep learning is mainly focused on catching this structure.

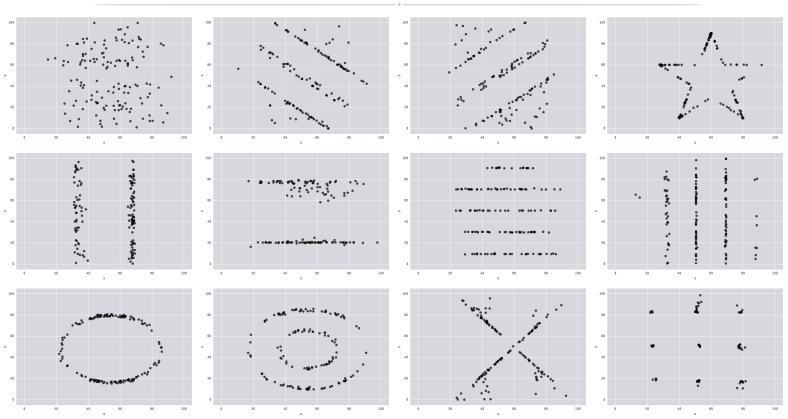
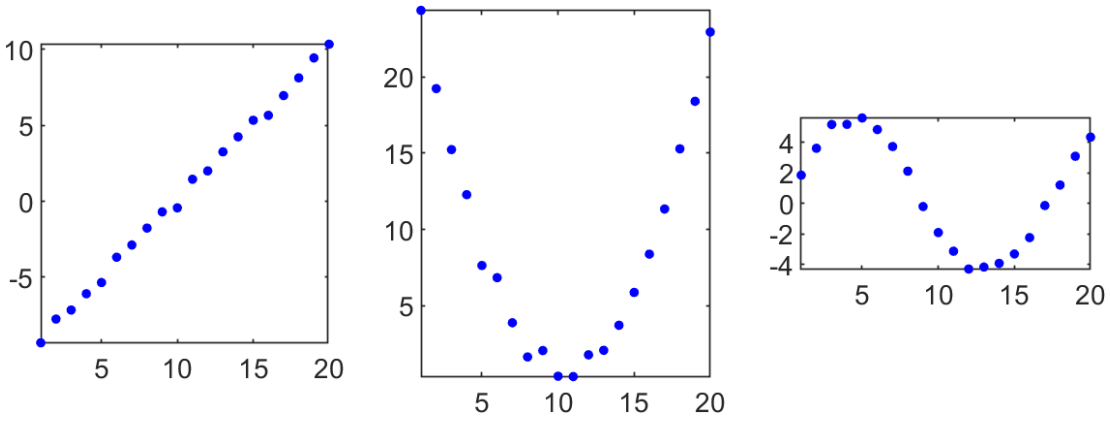


Figure 1.3: Datasets with the same statistics.

Nevertheless, it will not always be easy to visualize the data, since data can be *high-dimensional*, we may have just *implicit* access to data (e.g. latent spaces) or *no physical access* at all.

1.1 Models for describing the data

Learning is about *describing* data, or more specifically, describing the *process*, or *model*, that yields a given output from a given input.



$y = ax + b$ $y = ax^2 + bx + c$ $y = ax^3 + bx^2 + cx + d$

Our model might use *prior knowledge* on the data. For example, in the third plot, we might know a priori that the data actually comes from a periodic process $y = a\sin(x) + bx + c$. In general, one should look at the world, identify what knowledge he has about it, and use this knowledge to construct his model. Some forms of prior knowledge are for example

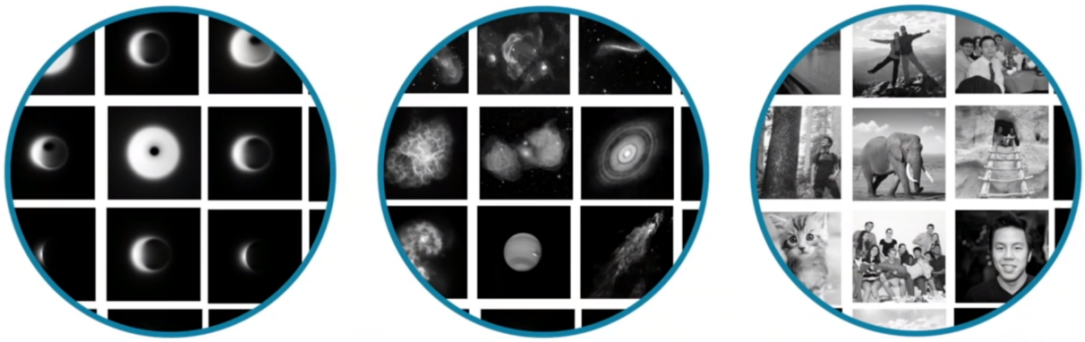
- *data distribution*, e.g. we know that the data come from an oscillatory process, thus they must follow a periodic distribution;
- *energy function*, e.g. we know that the data come from an energy-minimizing process;
- *constraints*, e.g. we know that all the data come from a fixed camera, of which we know all the intrinsic parameters;
- *invariances*, e.g. we work on pictures of snowflakes, which are rotationally symmetric;
- *input-output examples* (data prior).

All these encode, to different extents, some expected behaviour.

1.1.1 Reliability of the prior

Black hole imaging

The problem of imaging the black hole consisted of reconstructing an image from a sparse set of spectral measurements. It is an ill-posed inverse problem, since an infinite number of possible images explain the data. Thus, optimization needed to realy heavily on priors: the problem became finding an explanation that respected prior assumptions about the visual universe while still satisfying the observed data.



black holes
unreliable

astronomy

everyday

If we had to choose between the datasets in figure to learn reasonable priors regarding the visual universe, one would be tempted to use black hole images; the problem is that we don't have any black hole image, the ones we have come from artistic represention or simulations, so we would be introducing a bias to obtain what we already expect. The astronomy dataset would be a good idea, while everyday images are probably too much visually different from the black hole. Nevertheless, the learning process given to the laboratories across the world was so reliable that even if feeded with different datasets, it yielded almost the same image.

Fairness

AI is objective only in the sense of learning what human teaches. The data provided by humans can be highly biased.
Socially unfair behaviours can rise when the given data contains social prejudices; for example, the *Xing* social network gave male persons a better rank than women having better observed scores.
The first thing one should do is be aware of this risk, since it is quite challenging to avoid. Some possible causes are:

- *Skewed sample*: a tiny initial bias grows over time, since future observations confirm prediction. Example: police intercept crime more densely in areas they watch.
- *Tainted examples*: data produced by a human decision can be biased, and the bias is replicated by the system. Example: the Xing ranking system seen before.
- *Sample size disparity*: training data for a minority group is much less than the majority group.

In general, assessing data and prior reliability is crucial for any learning-based system.

1.1.2 Explaining the data

Learning is about discovering a *map* from input to output. Finding a model explaining the data means determining the map. The key assumption that we are going to use is that data has an *underlying structure*; however, it is very infrequent that this structure can be captured by a simple expression.

Choosing a representation The same data can be described in different ways, take for example the following curve

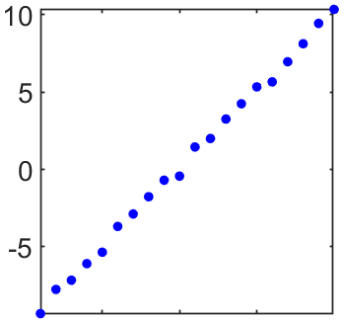


Figure 1.4: A linear curve.

this curve can be represented as a *function* with 2 weights

$$y = ax + b \tag{1.1}$$

or as a *parametric curve* with 4 weights:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_x \\ a_y \end{pmatrix} t + \begin{pmatrix} b_x \\ b_y \end{pmatrix}. \tag{1.2}$$

in the former, each datapoint is the output of a function of one single number x; in the latter, each point is modeled as a pair of numbers (x, y) linked to a variable t with a linear model over x and a linear model over y. In general, a parametric curve is a normal curve where we choose to define the curve's x and y values in terms of another variable. In this example, $a_x = 1, b_x = 0$ so the value for x is completely determined by the value of t, while the values of a_y, b_y are those of a, b of the previous formula.

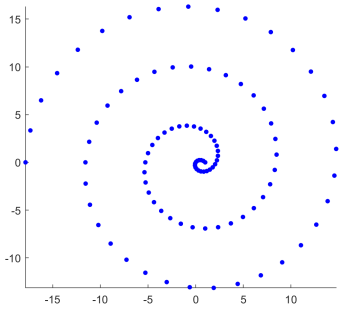


Figure 1.5: A spiral-shape dataset.

In fig. 1.5 instead, it seems that there is no way to describe the given dataset with a function, but we can easily do it with a parametric curve

$$\begin{pmatrix} x \\ y \end{pmatrix} = a \begin{pmatrix} cos(t) \\ sen(t) \end{pmatrix} (a - t). \tag{1.3}$$

Actually, the dataset can also be described with a function, by switching to polar coordinates

$$r = a\theta \tag{1.4}$$

which is even linear. In general, there is a trade-off between the number of weights and the simplicity of the model.

1.2 The curse of dimensionality

Until now we have seen examples in 1 or 2 dimensions but in practice we will often deal with higher dimensional cases, like images.
Consider a greyscale image (each pixel value is $\in [0,1]$), of width w and height h, it will have a total of wh dimensions; this image can be considered a point in a wh-dimensional space. A dataset of such images is a point cloud in $\mathbb{R}^{wh}$.

$\in \mathbb{R}^{w \times h} \cong \mathbb{R}^{wh}$

For example, a $\sim$1 megapixel photo (grayscale) has $\sim 10^6$ dimensions. Often, the question to ask is whether all these dimensions are significant.
Let's now delve into an explanation of the so-called *curse of dimensionality*; for simplicity, consider 1×1 images consisting of one single pixel. Since there is only one dimension, we can put all these images along the real axis, sorting them by value.

Figure 1.6: 1×1 images on the real axis.

Now consider 2×1 images, and place them as points in the 2-dimensional space with respect to their value.

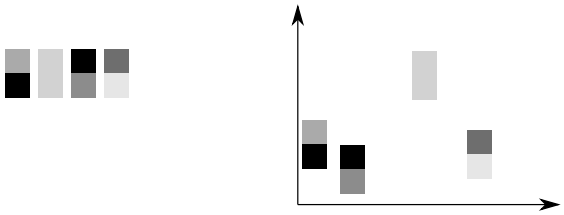


Figure 1.7: 2×1 images on the real axis.

If we do this for 3×1 images, we can see that as the dimensionality grows, the points in the space become more and more sparse; increasing dimension increases the sparsity of the point cloud.

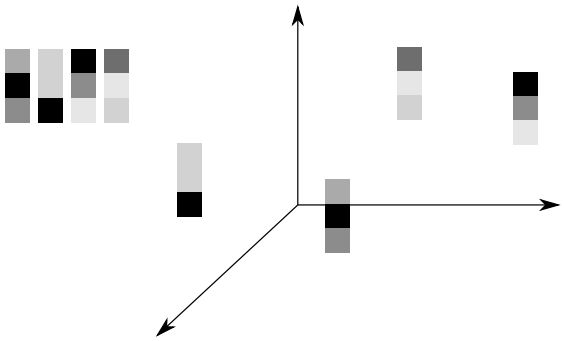


Figure 1.8: 3×1 images on the real axis.

A dataset of natural images will be extremely sparse in $\mathbb{R}^{wh}$, since each region of the ambient space will be observed very infrequently, expecially if we have few images. As we add new images, each one will not be very likely to fall close to the previous one. It can be proved that as we increase dimensionalities all points (each point is an image of the dataset) will eventually become equally spaced from the others.
If our dataset consists of equally spaced points, no meaningful structure will emerge from the dataset, and therefore there won't be anything interesting to learn about it. We would need to increase the number of data points (so the images of the dataset) exponentially in the number of dimensions. If n data points cover well the space of 1-dimensional images, then n^d data points are required for d-dimensional images.

1.2.1 Decrease the dimensions

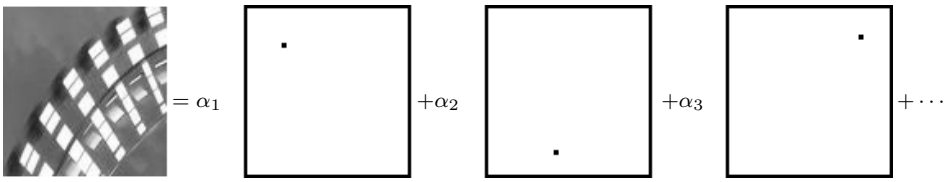
Features

Assume each data point $x \in \mathcal{D} \subset \mathbb{R}^n$ is the result of a synthesis, or generation, process:

$$\sigma : F \to x \tag{1.5}$$

which takes a set of features F and composes them to form x. Deep learning is about discovering F from x and simultaneously discovering σ.

Example 1.1. *An image $x \in \mathbb{R}^{w \times h}$ is composed by pixels.*
If each pixel of x is a feature, then σ simply sums them up:



In this case, the feature space *F is spanned by individual pixels. Each feature (each pixel) represents a dimension. This can be expressed concisely as*

$$x = \sigma(F) = \sum_{f_i \in F} \alpha_i \cdot f_i \tag{1.6}$$

where α_i are the weights *in the representation of x. In this* ***particular case****, the feature space is a* vector space *and σ is* linear*.*

Having one feature per pixel is extremely wasteful; in general, the curse of dimensionality occurs when

$$features \gg observations$$

In cases like this, we may want to find a different representation which doesn't fall in the curse of dimensionality, so we are basically asking what really characterizes our image.

Example 1.2. *The following image may be represented as a combination of these three features:*

or we may try to use even fewer features: the image may be represented using the first square, representing the white square as a modified version of the black box with different size and color and the black line as a degenere case.

In general, the transformation σ acts *nonlinearly* on the features. The output of σ is called an *embedding* of the data point. For the data point $x \in \mathcal{D} \subset \mathbb{R}^n$, the *embedding space* is $\mathbb{R}^n$.
What we are actually doing is not getting rid of the complexity, but hiding it in σ.
In deep learning, σ will be a deep neural network and the features will be the intermediate representation of the image; a trade-off between the number of features and the complexity of σ must be decided.

Intrinsic invariances In general, a given data point admits many possible embeddings.

Example 1.3. *A sheet lives naturally in $\mathbb{R}^2$, but is usually embedded in $\mathbb{R}^3$.*

Figure 1.9: Three different embeddings of the same object.

Even if the embeddings look totally different, distances are preserved in all of them.

In general, the challenge will be to discover what *intrinsic* properties are preserved; these properties characterize the data.

Latent features In the general case:

- features are not necessarily *localized* in space, and
- features are not necessarily *evident* in the embedding.

We thus talk about *latent* features, which are characterizing properties of the image that are hidden; more examples may be necessary to extract this kind of features. The problem is that we only have direct access to the embedding.

Example 1.4. *It is obvious for an human that the only difference among the various images in fig. 1.10 is that the light source is moving around the face. However, this is may not be obvious for a learning model. Let's say we want to design a model which, taken an image, relights it from different positions. The model can be parametrized with only 4 parameters,* (x, y, z) *for the light position and maybe another parameter for the intensity.*

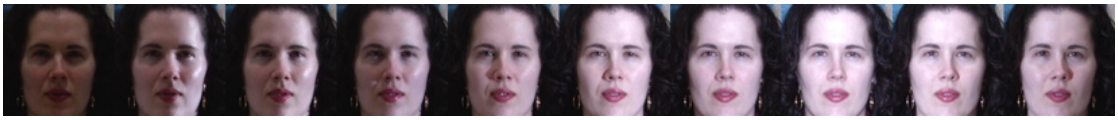


Figure 1.10: Example of a latent feature: directional illumination.

In general, discovering latent features involves discovering:

- the "true" embedding space for the data, and
- the *transformation* between the two spaces.

We would like to discard the *non-informative* dimensions from the data.

Dimensionality Even just discovering the intrinsic dimensionality is a challenge by itself. Usually, it is specified by hand by whoever designs the learning model. In fig. 1.11 we can see a representation of the Hughes phenomenon; in the original experiment, a classification model was trained with different intrinsic dimensionalities (x axis) and the accuracy was recorded for each of these (y axis); each curve in the plot represents a dataset. As you can see, every curve reaches a local maximum with a certain dimensionality and then sees its accuracy decrease as the dimensionality gets bigger and bigger.

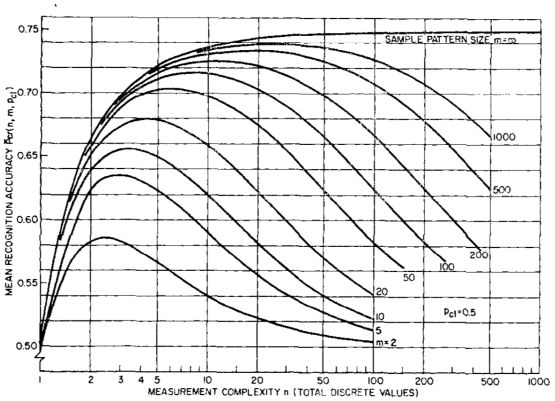


Figure 1.11: The Hughes phenomenon.

Finding a lower-dimensional embedding for some given data is a dimensionality reduction problem. Usually, this is done by nonlinear dimensionality reduction techniques.

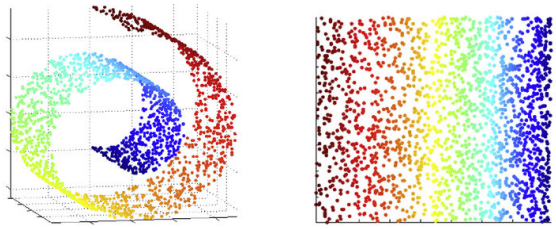


Figure 1.12: Swiss roll plot.

This class of problems is also called *manifold learning*. However it must not be confused with deep learning; the former only finds a lower-dimensional embedding for the data, while the latter finds patterns in the data, and also determines a map.
In deep learning the *manifold hypothesis* is assumed to hold; that is, the input data is assumed to live on some underlying non-Euclidean structure called a manifold.

Definition 1.1 (Manifold)**.** A manifold is a topological space that locally resembles Euclidean space near each point. More precisely, an n-dimensional manifold is a topological space with the property that each point has a neighborhood that is homeomorphic to the Euclidean space of dimension n.

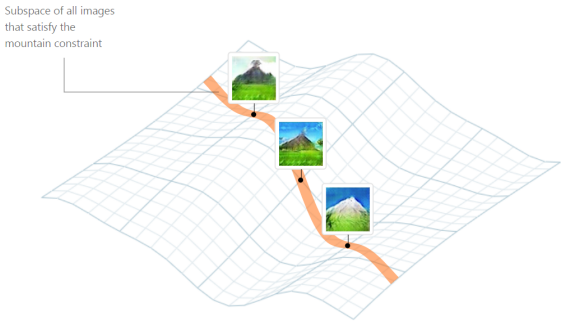


Figure 1.13: A manifold.

Task-driven features Speaking about features only makes sense if we are given a task to solve, for example, color in a deck of french cards is important only in some card games.
We are now ready to give a more comprehensive definition of deep learning.

Definition 1.2 (Deep learning)**.** Deep learning is a task-driven paradigm to extract patterns and latent features from given observations.

It must be said nevertheless that features are not always the focus of deep learning; rather, they are usually instrumental for the given task.

Chapter 2

Linear regression, convexity and gradients

Let us recap briefly what is the general setting: in deep learning, we deal with *highly parametrized* models, usually in the order of millions or even hundreds of millions of parameters, and these models are called *deep neural networks*. A deep neural network models some function f, possibly nonlinear, parametrized by parameters Θ. In general, given a data point x in input to the neural network, this returns an output y:

$$f_{\boldsymbol{\Theta}}(\mathbf{x}) = \mathbf{y} \tag{2.1}$$

A neural network takes the form of a composition of multiple simpler blocks each of which has a predefined structure (*e.g.* one block might be modelling a linear map). The structure is chosen by who designs the neural network, so this is not something that we solve for, but is fixed during the design of the neural network.
Each block is defined in terms of unknown parameters Θ and the collection of the parameters of all the blocks are the parameters of the entire network.

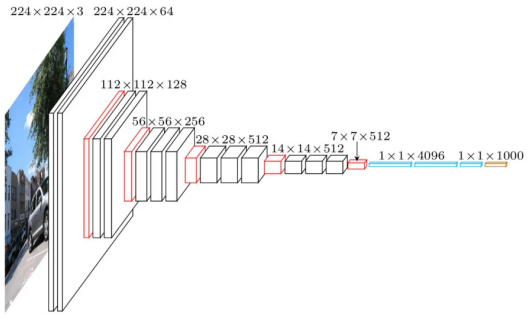


Figure 2.1: Example of a neural network structure

We usually have access to few data points X and a few data points Y, *i.e.* our training pairs of examples, and our task is to solve for the parameters Θ of the function $f_{\boldsymbol{\Theta}}$ that *is most likely* to have produced Y from X, i.e. we have to *infer* the function f from $\{X, Y\}$. Finding the values of the function parameters $\boldsymbol{\Theta}$ is called *training*.

In order to do this, we need to define some criterion with which we can say that the *learned function* $f_{\mathbf{\Theta}}$ is more or less likely to represent *true function* f. This is usually done by defining some energy function that depends on the produced output $\hat{Y} = f_{\mathbf{\Theta}}(X)$, and so indirectly on the parameters $\mathbf{\Theta}$ (since these influence the learned function $f_{\mathbf{\Theta}}$), that we usually call *loss function*, which we want to minimize. Finding the parameters that minimize the loss function will be done using an optimization procedure that requires computing gradients, so it involves what we call *backpropagation*, *i.e.* the process of computing derivatives of all the functions involved in the network.

2.1 Linear Regression

The simplest non-trivial case for a learning model is going to be a linear model called *linear regression*.

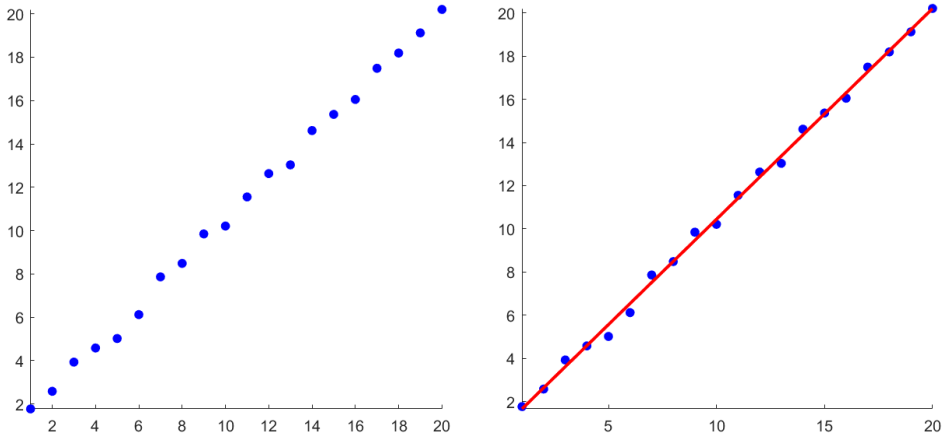


Figure 2.2: Linear regression problem.

Given some data points we assume that these data points come from a linear process, *e.g.*

$$y_i = ax_i + b$$

(note that this is not the only possible choice) and we want to look for the values of the parameters a and b. We have to consider an additive correction, that we call *noise*, since we don't expect the line to fit exactly every datapoint.

$$y_i = ax_i + b + noise \tag{2.2}$$

In the linear regression setting this is ignored, as fitting the line perfectly to every datapoint would lead us to *overfitting*, i.e. poor *generalization*. We will see these concepts more in depth in the following chapter.
Linear regression has the following properties:

- it is linear plus a bias b (ignores the noise);
- has parameters $\Theta = \{a, b\}$;
- needs a labeled dataset of n pairs (x_i, y_i), where x_i's are called the regressors.

So we can write the model as:

$$f_\Theta(x_i) = y_i \tag{2.3}$$

Once we are given values for a and b we can construct a mapping such that given some new input we can apply the function learned and get some new output.
But who gives us these values? As we have mentioned previously, this is achieved by an optimization process of a proper loss function, so we need to define one. Often a loss function encodes some notion of *error*, measuring the distance between the true value y_i from what our f_Θ predicts for the value x_i, and we want that difference to be close to 0. Usually the notion of error that we use for the linear regression setting is the *Mean Squared Error (MSE)*:

$$\epsilon = \min_{a,b\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n} \left(y_i - f_\Theta(x_i)\right)^2. \tag{2.4}$$

We are looking for the parameters values that when plugged into the function f, give the minimum possible error ϵ. This is a *minimization problem* and we don't care about the minimum value but rather we are interested in the values $\bar{\Theta} = \bar{a}, \bar{b}$ for the parameters (minimizers) that make the error reach that minimum value, so the problem might be formalized as:

$$\Theta^* = \operatorname*{argmin}_{a,b\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n} \left(y_i - f_\Theta(x_i)\right)^2. \tag{2.5}$$

As we will see, not all minimization problems have a *closed-form* solution (i.e. admit a formula to obtain $\bar{\Theta}$). However, when f_Θ is linear, the problem becomes a so called *least-square approximation* problem, for which, as we will see, such a closed-form solution does exist.
Note that to look for the minimizer we can ignore the constant factor $1/n$, so we can remove it:

$$\epsilon = \min_{a,b\in\mathbb{R}} \sum_{i=1}^{n} \left(y_i - f_\Theta(x_i)\right)^2 \tag{2.6}$$

The error criterion, as mentioned previously, is also called *loss function*, usually denoted by ℓ_Θ:

$$\ell_\Theta\left(\{x_i, y_i\}\right) = \sum_{i=1}^{n} \left(y_i - f_\Theta(x_i)\right)^2 \tag{2.7}$$

so the *MSE* becomes:

$$\epsilon = \min_{\Theta} \ell_\Theta\left(\{x_i, y_i\}\right) \tag{2.8}$$

Remember that the loss is defined on the entire dataset not on just one datapoint.
In summary, we have access to examples $\mathbf{x}_i$ and $\mathbf{y}_i$, we fix the structure of the function f and we need to solve for the parameters Θ. To solve for Θ, we define a loss function ℓ_Θ that depends on the examples $(\mathbf{x}_i, \mathbf{y}_i)$ and the parameters, and we minimize it, i.e. we find the parameters Θ^* that make it minimum.

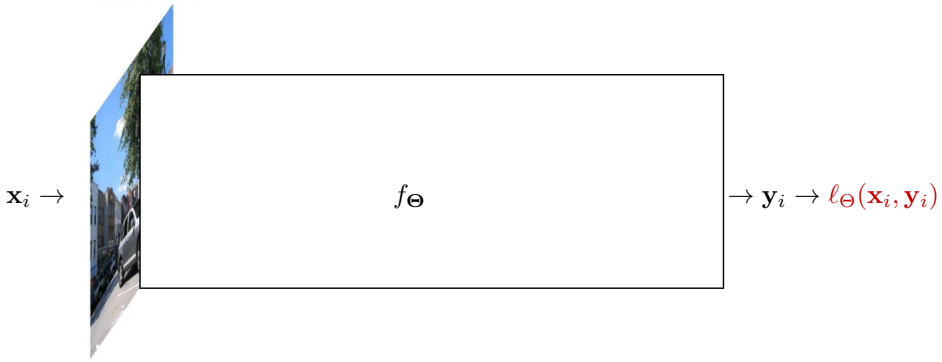


Figure 2.3: Task of a deep neural network.

(Note that the $\mathbf{x}_i$'s $\mathbf{y}_i$'s and the parameters Θ could be higher dimensional vectors).
In linear regression we fix f to be linear and ℓ to be quadratic.

2.2 Convexity

We need to solve the general minimization problem:

$$\epsilon = \min_{\Theta} \ell(\Theta) \tag{2.9}$$

In particular, we are interested in the minimizer Θ. Finding minimizers for a general loss function is an open problem of the research area called *optimization*. The optimization method depends on the specific property of the loss function, and, as we will see, there might sometimes be constraints on the parameters; nevertheless, we will mostly deal with unconstrained problems.
There are some classes of functions that are easier to minimize (or maximize) and the easiest set of functions is the set of convex functions, defined by *Jensen's inequality*:

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha) f(y), \ \forall x, y \text{ and } \alpha \in [0,1] \tag{2.10}$$

The inequality states that the convex transformation of a mean is less than or equal to the mean applied after convex transformation; for two points, it says that the secant line of a convex function lies above the graph of the function.

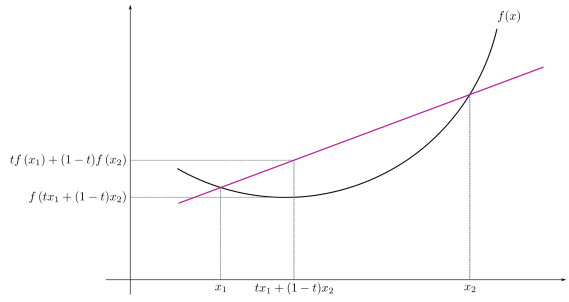


Figure 2.4: Jensen's inequality generalizes the statement that a secant line of a convex function lies above the graph.

Why are these type of functions the easiest to minimize? Looking at the graph above, we can intuitively appreciate how there always exists a *unique* minimum. If you recall some high school calculus you also know that the point where such minimum is achieved is the (again, unique) point where the *derivative* of the function is zero.
Therefore, an important assumption that we often make is for the loss function ℓ to be differentiable, so we can compute its derivative $\frac{\mathrm{d}\ell}{\mathrm{d}x}$ at all points x.
To explain this formally, let us write a different form of eq. (2.10):

$$f(x + \alpha(y - x)) \leq (1 - \alpha)f(x) + \alpha f(y), \qquad \forall x, y \text{ and } \alpha \in (0, 1) \tag{2.11}$$

doing some algebraic manipulation we have:

$$\frac{f(x + \alpha(y - x))}{\alpha} \leq \frac{(1 - \alpha)f(x) + \alpha f(y)}{\alpha} \tag{2.12}$$

$$\frac{f(x + \alpha(y - x))}{\alpha} \leq \frac{f(x)}{\alpha} - f(x) + f(y) \qquad \text{(expanding the product } (1 - \alpha)f(x))$$

$$\frac{f(x + \alpha(y - x)) - f(x)}{\alpha} + f(x) \leq f(y). \tag{2.13}$$

Now, we take the limit

$$\lim_{\alpha \to 0} \frac{f(x + \alpha(y - x)) - f(x)}{\alpha} + f(x) \leq f(y) \tag{2.14}$$

and notice how it resembles the definition of the derivative of a function, i.e. the limit of the difference quotient:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \tag{2.15}$$

To complete the expression we need a factor $(y - x)$:

$$\lim_{\alpha \to 0} \frac{f(x + \alpha(y - x)) - f(x)}{\alpha(y - x)}(y - x) + f(x) \leq f(y) \qquad \forall x, y. \tag{2.16}$$

Since $(y - x)$ is just a scalar we can move it out from the limit, so the entire limit represents the derivative of f:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x}(y - x) + f(x) \leq f(y) \qquad \forall x, y. \tag{2.17}$$

Notice that the left hand side of the inequality is just the 1-st order Taylor approximation of f at point x, *i.e.* the best possible approximation of f as a linear function at a given point x and this would be the line that is tangent to the curve at that point.

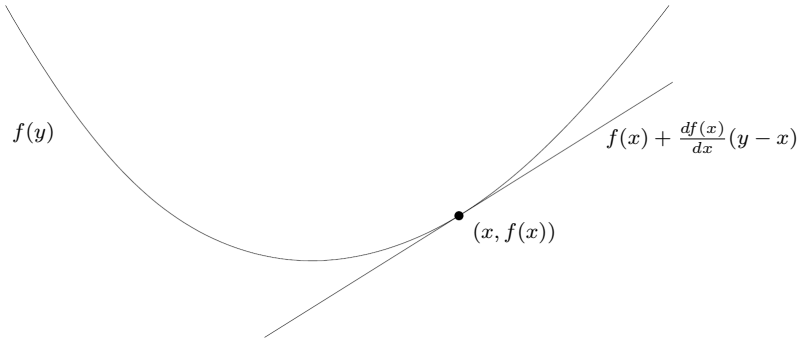


Figure 2.5: Taylor approximation.

Notice also that the slope of the line is given by the derivative, and that a flat line happens when the slope (the derivative) of the line is zero. Setting $\frac{\mathrm{d}f}{\mathrm{d}x} = 0$ we have that:

$$f(x) \leq f(y) \qquad \forall y \tag{2.18}$$

and therefore x is a global minimizer of f, as mentioned previously.
In general, if we want to find a minimizer for a convex function f we just need to compute its derivative $\frac{\mathrm{d}f}{\mathrm{d}x}$, set it to zero and solve for x; then as we have shown the point x will satisfy eq. (2.18) and hence will be the global minimizer of the function.

2.3 Gradients

Usually in deep learning we don't deal with univariable, scalar-valued functions like we have seen before, i.e. functions $f : \mathbb{R} \to \mathbb{R}$, but with multivariable, often vector-valued functions, i.e. functions $f : \mathbb{R}^n \to \mathbb{R}$, often called scalar fields or functions $f : \mathbb{R}^n \to \mathbb{R}^n$, often called vector fields.
Let us concentrate on scalar fields, since the more troubling part of moving from $\mathbb{R}$ to $\mathbb{R}^n$ is when this happens in the *domain* of the functions, not in its *codomain*, i.e. when we have functions with multiple arguments. In fact, vector-valued functions are simply vectors whose components are scalar-valued functions, i.e. a stack of scalar fields. However, when a function involves multiple variables things are not so simple. This is the case for the notion of derivative: for functions of a single variable, derivatives are straight-forward, since there is only one variable that can cause a change in the function variable; that is not true in the multivariable case.
For this reason, in this setting we need to replace (generalize) the notion of derivative with the notion of *gradient*,

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \tag{2.19}$$

which is the vector of *partial derivatives* of f, where $\mathbf{x}$ is a high-dimensional vector whose components are the multiple variables of the function.
Partial derivatives are just regular derivatives performed when all the variables of the functions are freezed but the one under consideration. For example consider $f(x, y) = 2x^2y + sin(x)y^2$:

$$\frac{\partial f}{\partial x} = 4xy + ycos(x) \qquad \frac{\partial f}{\partial y} = 2x^2 + 2sin(x)y \tag{2.20}$$

we have freezed y in the first partial derivative (treating it as a constant) and differentiated f with respect to x like in the univariable case, and the reverse for the second partial derivative.
Now we can recover the notion of convexity, but now it involves vectors:

$$f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1-\alpha) f(\mathbf{y}). \tag{2.21}$$

The global optimality condition becomes

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{0} \implies f(\mathbf{x}) \leq f(\mathbf{y}),\ \forall \mathbf{y} \in \mathbb{R}^n, \tag{2.22}$$

with the gradient effectively replacing the derivative.
Let's ponder for a moment what this replacement means. Recall that the derivative informs us of the rate of change of the function, i.e. tells us how to change the independent variable to

increase or decrease its value. In the univariable setting the independent variable lies on the real axis, in which there is only one possible *direction*. The *sign* of the derivative tells us the *orientation* that the independent variable has to follow in order for the function to increase. This is not true any longer in the multivariable case. There are multiple directions in which the independent variable $\mathbf{x}$ (now a vector) can be shifted, and the function's value can be affected in different ways along different directions. This notion is encoded by the *directional derivatives*:

$$\frac{\partial f}{\partial \mathbf{v}} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h}. \tag{2.23}$$

However, since $\mathbf{x}$ and $\mathbf{v}$ are vectors, if the function is differentiable to know how the function will change along any direction of change $\mathbf{v}$ it suffices to know how the function changes along the directions of the basis of its vector space. In fact it can be shown that it is

$$\frac{\partial f}{\partial \mathbf{v}} = \nabla f(\mathbf{x})^\top \mathbf{v}. \tag{2.24}$$

From this expression we can appreciate how the gradient represents the direction of *steepest ascent* of f at point $\mathbf{x}$ (note that the direction of increase is on the domain of f).

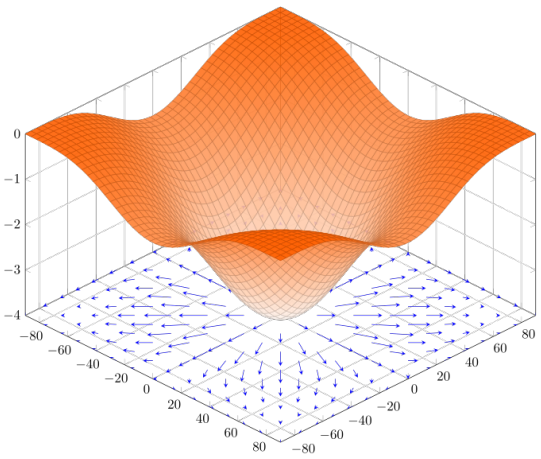


Figure 2.6: The gradient points towards the direction of steepest ascent.

In fact, it is

$$\frac{\partial f}{\partial \mathbf{v}} = \nabla f(\mathbf{x})^\top \mathbf{v} \overbrace{=}^{\text{by definition of dot product}} \|\nabla f(\mathbf{x})\| \|\mathbf{v}\| \underbrace{\cos(\theta)}_{\text{angle between the gradient vector and the direction vector}} \tag{2.25}$$

but the direction vector is a unit vector hence its norm is 1, therefore we end up with

$$\frac{\partial f}{\partial \mathbf{v}} = \|\nabla f(\mathbf{x})\| \cos(\theta) \tag{2.26}$$

which means that the directional derivative along a given direction $\mathbf{v}$ is proportional to the norm (or magnitude, or length) of the gradient vector times the cosine of the angle in between. In particular, it is maximum when $\cos(\theta) = 1$ which is when $\theta = 0$ therefore the two vectors have the same direction, and this means that indeed the gradient vector encodes the direction in which the directional derivative is maximum, i.e. the direction of steepest ascent.

On the other hand the *length* of the gradient vector encodes how much the function increases in the direction of steepest ascent. However, although one can intuitively understand what we mean by length, we have not defined it.
Indeed, when defining what a vector space is, we made no reference to any concept of length, or direction, or orientation whatsoever. However, the Euclidean space $\mathbb{R}^n$ is not just a vector space, but is equipped with additional structures; in particular, it is equipped with a *metric*, and is thus a *metric space*, i.e. it has a notion of *distance* between two points, that actually gives the name to the whole space since it is called the *Euclidean distance*, which is the one we are used to.
A metric or distance is a function $d(\cdot,\cdot) \to \mathbb{R}$ that tells us how distant two points of a metric space are from each other. One can define any distance, for instance the Euclidean distance is a particular instance of a more general class of distance functions, called L_p distance, which expressed in matrix notation, is:

$$d(\mathbf{x},\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|_p \triangleq \left(\sum_{i=1}^{k} |x_i - y_i|^p\right)^{\frac{1}{p}}. \tag{2.27}$$

We can appreciate how setting $p=2$ in the definition above we recover the usual definition of Euclidean distance, therefore it is also sometimes referred to as L_2 distance:

$$d(\mathbf{x},\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|_2 = \left(\sum_{i=1}^{k} |x_i - y_i|^2\right)^{\frac{1}{2}} = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \cdots + (x_n-y_n)^2}. \tag{2.28}$$

Once we have a space equipped with a metric, we can measure distances between vectors from that space. With this tool we can define a notion of *length*, or *norm* of a vector as the distance of the vector from the origin:

$$\|\mathbf{x}-\mathbf{0}\|_2 = \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{k} |x_i|^2} = \sqrt{\mathbf{x}^\top \mathbf{x}}; \tag{2.29}$$

we now have a *normed* space, of which the Euclidean space is the most known instance.
So, going back to linear regression, we have that:

$$\boldsymbol{\Theta}^* = \underset{\boldsymbol{\Theta}\in\mathbb{R}}{\operatorname{argmin}}\, \ell(\boldsymbol{\Theta}) \tag{2.30}$$

where

$$\ell(a,b) = \sum_{i=1}^{n} (y_i - a x_i - b)^2 \tag{2.31}$$

we can see that ℓ is a convex function, since a quadratic function is convex and so is the sum of quadratic functions, so we can find the parameters that globally minimize the loss function by setting the gradient equal to $\mathbf{0}$

$$\begin{aligned}
\nabla_{\boldsymbol{\Theta}} \ell(\boldsymbol{\Theta}) &= \nabla_{\boldsymbol{\Theta}} \sum_{i=1}^{n} (y_i - a x_i - b)^2 = \sum_{i=1}^{n} \nabla_{\boldsymbol{\Theta}} (y_i - a x_i - b)^2 && \text{(by linearity of gradient)} \\
&= \sum_{i=1}^{n} \nabla_{\boldsymbol{\Theta}} (\cancel{y_i^2} + a^2 x_i^2 + b^2 - 2 a x_i y_i - 2 b y_i + 2 a b x_i) && (\nabla_{\boldsymbol{\Theta}} y_i^2 = 0) \\
&= \sum_{i=1}^{n} \begin{pmatrix} 2 a x_i^2 - 2 x_i y_i + 2 b x_i \\ 2 b - 2 y_i + 2 a x_i \end{pmatrix} && (2.32)
\end{aligned}$$

where the first element of the vector is the partial derivative with respect to a and the second element is the partial derivative with respect to b

$$\nabla_{\boldsymbol{\Theta}} \sum_{i=1}^{n} (y_i - ax_i - b)^2 = \begin{pmatrix} \sum_{i=1}^{n} 2ax_i^2 - 2x_i y_i + 2bx_i \\ \sum_{i=1}^{n} 2b - 2y_i + 2ax_i \end{pmatrix} \tag{2.33}$$

Now we can set the gradient equal to zero and solve the system of two linear equation (linear in a and b)

$$\begin{pmatrix} \sum_{i=1}^{n} ax_i^2 + bx_i - x_i y_i \\ \sum_{i=1}^{n} ax_i + b - y_i \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{2.34}$$

Similarly, in matrix notation we can write all the equations $y_i = ax_i + b$ at once to make the linearity w.r.t. a, b evident:

$$\underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}}_{\mathbf{y}} = \underbrace{\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}}_{\mathbf{X}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\boldsymbol{\theta}} \tag{2.35}$$

Then, the MSE can be expressed as the L_2 distance:

$$\begin{aligned} \ell(\boldsymbol{\theta}) &= \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 && (2.36)\\ &= (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^\top (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) && (2.37)\\ &= \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{X}\boldsymbol{\theta} - (\mathbf{X}\boldsymbol{\theta})^\top \mathbf{y} + \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\theta} && (2.38)\\ &= \mathbf{y}^\top \mathbf{y} - 2\mathbf{y}^\top \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\theta} && (2.39) \end{aligned}$$

where in eq. (2.39) we used the fact that both $\mathbf{y}$ and $\mathbf{X}\boldsymbol{\theta}$ are vectors, and the dot product between vector is commutative.
Setting the gradient w.r.t. $\boldsymbol{\theta}$ to zero we get

$$\begin{aligned} -2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X}\boldsymbol{\theta} &= \mathbf{0} && (2.40)\\ \mathbf{X}^\top \mathbf{X}\boldsymbol{\theta} &= \mathbf{X}^\top \mathbf{y} && (2.41)\\ \boldsymbol{\theta} &= (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} && (2.42) \end{aligned}$$

In the general case, we have higher-dimensional datapoints:

$$\mathbf{y}_i = \mathbf{A}\mathbf{x}_i + \mathbf{b} \quad \text{for } i = 1, \ldots, n \tag{2.43}$$

Stacking all data points into matrices $\tilde{\mathbf{X}} = \begin{pmatrix} | & | \\ \mathbf{x}_1 & \mathbf{x}_2 \cdots \\ | & | \end{pmatrix}$ and $\mathbf{Y}$, we get:

$$\underbrace{\begin{pmatrix} y_{11} & \cdots & y_{1d} \\ y_{21} & \cdots & y_{2d} \\ \vdots & & \vdots \\ y_{n1} & \cdots & y_{nd} \end{pmatrix}}_{\mathbf{Y}^\top} = \underbrace{\begin{pmatrix} x_{11} & \cdots & x_{1d} & 1 \\ x_{21} & \cdots & x_{2d} & 1 \\ \vdots & & \vdots & \vdots \\ x_{n1} & \cdots & x_{nd} & 1 \end{pmatrix}}_{\mathbf{X}^\top := (\tilde{\mathbf{X}}^\top | \mathbf{1})} \underbrace{\begin{pmatrix} a_{11} & \cdots & a_{1d} \\ \vdots & & \vdots \\ a_{d1} & \cdots & a_{dd} \\ b_1 & \cdots & b_d \end{pmatrix}}_{\boldsymbol{\Theta}} \tag{2.44}$$

where to get the matrix $\boldsymbol{\Theta}$ we just transposed the vector $\mathbf{b}$ and put it below the matrix $\mathbf{A}$.

According to eq. (2.44), for each output data point $\mathbf{y}_i$ we have:

$$\underbrace{\begin{pmatrix} y_{i1} \\ \vdots \\ y_{id} \end{pmatrix}}_{\mathbf{y}_i} = \begin{pmatrix} \sum_{j=1}^{d} a_{j1} x_{ij} + b_1 \\ \vdots \\ \sum_{j=1}^{d} a_{jd} x_{ij} + b_d \end{pmatrix} \tag{2.45}$$

So, the *MSE* becomes

$$\ell(\mathbf{\Theta}) = \|\mathbf{Y}^\top - \mathbf{X}^\top \mathbf{\Theta}\|_2^2 = \mathrm{tr}(\mathbf{Y}^\top \mathbf{Y}) - 2\mathrm{tr}(\mathbf{Y}\mathbf{X}^\top \mathbf{\Theta}) + \mathrm{tr}(\mathbf{\Theta}^\top \mathbf{X}\mathbf{X}^\top \mathbf{\Theta}) \tag{2.46}$$

Chapter 3

Nonlinear models, overfitting and regularization

Having more data allows us to improve our predictions but can even confute some assumptions, *e.g.* in the following figure, having more data confutes the assumption that the distribution of data behaves linearly

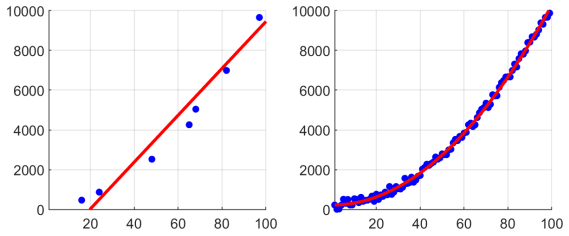


Figure 3.1: More data confutes the linear assumption.

Therefore, we face some key questions:

- How to select the right distribution?
- How much data do we need?
- What if the correct distribution does not admit a simple expression?

3.1 Nonlinear models

After the linear model, the simplest thing is something that follows a polynomial model, and this is called *polynomial regression*, expressed with the following equation:

$$y_i = b + \sum_{j=1}^{k} a_j x_i^j, \text{ for all data points } i = 1, \ldots, n \tag{3.1}$$

parametrized by b and the a_js.

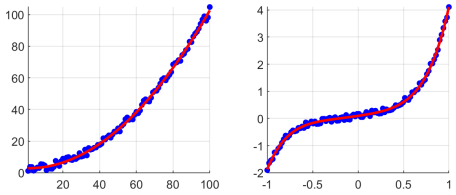


Figure 3.2: Polynomial regression.

As we increase the degree of the polynomial, also the number of the parameters to estimate will grow. Moreover, if we are dealing with data which is high dimensional then the number of the parameters is even bigger. More data are needed to make an informed decision about the order of the polynomial.
Despite the name, polynomial regression is still linear in the parameters, but polynomial with respect to the data; for this reason instead of polynomial regression we should call it *linear regression with polynomial features*. So the polynomial regression can be expressed in matrix notation by rewriting eq. (3.1) for every component y_i as:

$$\underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}}_{\mathbf{y}} = \underbrace{\begin{pmatrix} x_1^k & x_1^{k-1} & \dots & x_1 & 1 \\ x_2^k & x_2^{k-1} & \dots & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^k & x_n^{k-1} & \dots & x_n & 1 \end{pmatrix}}_{\mathbf{X}} \underbrace{\begin{pmatrix} a_k \\ a_{k-1} \\ \vdots \\ a_1 \\ b \end{pmatrix}}_{\boldsymbol{\theta}} \tag{3.2}$$

and $\mathbf{X}$ represents what we call the polynomial features. We can just apply the same exactly least-squares approach as we did for linear regression, but with the requirement that $k < n$ otherwise we do not have enough information to solve for all the parameters.

3.2 Polynomial fitting

How powerful is this model? Meaning, how many real functions f can it represent, given the proper values for the parameters? The *Stone-Weierstrass* theorem provides us with an aswer.

Theorem 3.1 (Stone-Weierstrass). *If f is continuous on the interval $[a, b]$, then for every $\epsilon > 0$ there exists a polynomial p such that $|f(x) - p(x)| < \epsilon$ for all x.*

Theorem 3.1 tells us that if we are given a real function f on some interval then there exists a polynomial p, whose degree we do not know, that *globally* (i.e. for every value of its domain x) approximates f to the desired accuracy, and the approximation is measured in the absolute error. But how to choose the degree of the polynomial? It is not a parameter to be learned, but rather an *hyperparameter*, that encodes a prior we establish, e.g. we expect the data to be distributed following a cubic-like polynomial. Let's see what happens by trying to fit polynomials of various degrees to the same data.

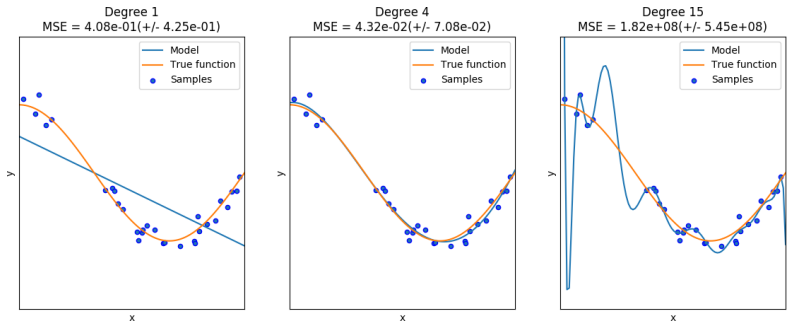


Figure 3.3: Fit data with different poly degree.

In fig. 3.3 we can see that with a 1-degree polynomial we can not represent the data distribution well: we are *underfitting*. With a 4-degree polynomial we get a very good representation, so we may think that going higher we could get an even better representation. Indeed, with a 15-degree polynomial we get a representation that fits all points, so the *MSE* for this polynomial is smaller than the 4-degree polynomial, although we can see immediately that the representation is not something that we desire. Why is that? Because it is very unlikely that the true function that has produced these data points looks like this learned function. We are *overfitting*.
These two phenomena that we observe in fig. 3.3 are found across all deep learning, not just polynomial regression models:

- **Underfitting**: not sufficiently fitting the data (large *MSE*) as it happens choosing degree one polynomial;
- **Overfitting**: we are "learning the noise" as it happens choosing degree fifteen polynomial.

From the notion of overfitting we see that adding complexity to a learning model is not necessarily a good thing because it could lead to overfitting and overfitting leads to bad generalization. So there will always be a trade-off between these two phenomena.

> **Note**: Remember that we are inferring a function (an item from an infinite-dimensional space) from a finite set of training samples, therefore there necessarily will be regions of the domain not covered by the samples, in which we have no clue of the behavior of the function (this is where the priors step in). Nonetheless, we would like for the learned function to approximate the true function well overall, not only on the training data, i.e. to be as *general* as possible, even if this means not fitting the training data perfectly.

There is a relatively easy way to detect whether we are doing underfitting or overfitting:

1. Separate the known data into two sets: the *training* set and the *validation* set;
2. Estimate the model parameters on the *training* set so as to minimize the loss function on the training data;
3. If the loss is large on the training set, then we are underfitting, since the model is not able to represent well enough the training data;
4. If the loss is small, then we *may* be overfitting. To check this, we take the *validation* set and compute the loss function on these new data;

5. If we get large loss on the validation set then we are overfitting, since the model is very good at representing the training data, but generalizes badly on unseen samples, hence the learned function cannot be a good global approximation of the true function.

In summary, underfitting is whenever we have large training error and large validation error, while overfitting is whenever we have small training error and large validation error.

k-fold cross-validation There are different mechanisms that defend us from underfitting and overfitting and the simplest way that we usually employ in practice is called *k-fold cross-validation*.
The key point of *k-fold cross-validation* is that we do not want to overfit on the validation set either. To avoid this we split our training set in k subsets which we call *folds*. Then we train on $k-1$ subsets and validate on the remaining 1, and repeat this task for each subset (*i.e.* k times). In this way, we have the *MSE* for each subset and we can take the average. If the model is getting a good score on the average over all the folds then we can say that the model is a good one, while if it does not get a good score then maybe we should change the model. For instance, in polynomial regression we can do *k-fold cross-validation* many times with different degrees and choose the run with the smallest average *MSE*.

Figure 3.4: k-fold cross validation.

So, we said that polynomials can approximate any continuous function by Theorem 3.1, but are not enough for many reasons:

- We can have much more complicated losses which are not expressed as *MSE*;
- Polynomial regression is not easy to regularize;
- Maybe we have some other knowledge, not just training data, that we would like to inject into the model;
- Some models produce intermediate features which are very helpful to address certain tasks;
- Flexibility, meaning ease to access or to experiment with;

3.3 Regularization

Sometimes our prior knowledge can be expressed in terms of an energy. For example in polynomial regression we may want to avoid large parameters in order to counteract overfitting and thus control the complexity of the learning model; for this purpose, we can sum to our minimization problem the squared *Frobenius norm* (a type of *matrix norm* that generalizes the L_2 norm

defined for vectors to matrices, so it is often referred to as simply L_2 norm, see Definition 3.1) of the parameters. In this case, the *regularizer* would be

$$\|\mathbf{\Theta}\|_F^2 = \left(\sqrt{\sum_i \sum_j |\Theta_{ij}|^2}\right)^2 = \sum_i \sum_j |\Theta_{ij}|^2 = tr(\Theta^\top \Theta). \tag{3.3}$$

Definition 3.1 (Entry-wise norms)**.** These norms treat an $m \times n$ matrix as a vector of size $m \cdot n$, and use one of the familiar vector norms. For example, using the p-norm for vectors, $p \geq 1$, we get:

$$\|A\|_{p,p} = \|\text{vec}(A)\|_p = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^p\right)^{1/p} \tag{3.4}$$

The special case $p = 2$ is the Frobenius norm, and $p = \infty$ yields the maximum norm.

In general, the regularizer is a function that depends on the parameters, over which it enforces some soft constraint by producing a scalar that is higher the more violated the constraint is. In this case, the soft constraint would be to bring the norm of the parameters as close to zero as possible. Of course this would make the model useless, so we need to balance the regularizer action with the representational power of the model. This is achieved by a scalar λ which trades off fidelity with respect to data with fidelity with respect to the regularizer, so

- $\lambda = 0$ means we just want to be as good as possible on the data, while
- $\lambda = \infty$ means that we do not care about the data, we are just asking for very small values for parameters.

The minimization problem thus becomes

$$\underset{\mathbf{\Theta}}{\text{argmin}} \underbrace{\ell_\Theta}_{\text{data term}} + \underbrace{\lambda}_{\text{trade-off}} \cdot \underbrace{\|\mathbf{\Theta}\|_F^2}_{\text{regularizer}} . \tag{3.5}$$

Adding a quadratic penalty to the loss is also known as *weight decay*. Other common names are *ridge* regularization, or *Tikhonov* regularization. Controlling the parameter growth is generally known as *shrinkage*, and weight decay is not the only way to do so: for instance the L_1 norm gives rise to *lasso* regularization, that has the following expression

$$\min_{\mathbf{\Theta}} \ell_\Theta + \lambda |\mathbf{\Theta}| \tag{3.6}$$

and induces *sparsity*, since every parameter will receive an equal "push" towards zero, regardless of their magnitude. On the other hand the ridge regularization induces a "push" that will be proportional to the actual magnitude of the parameter, so larger parameters will go faster to zero than smaller parameters, resembling an exponential decay (hence the name). With L_1 regularization, when a parameter is zero it will stay at zero, therefore achieving *sparsity* i.e. the model has some irrelevant parameters, and the matrix representing them is sparse.
Why does all of that happen? We have not yet seen how the training actually modifies the parameters to minimize the loss function, but we can anticipate that it has to do with gradients. This suffices to build an intuition on why the L_1 and L_2 behave this way. Let's take a single (scalar) parameter $\theta \in \Theta$, and let's compute the gradient (that reduces to a single partial

derivative) of the two regularizers with respect to θ.

$$\frac{\partial}{\partial\theta}\left(\lambda\|\mathbf{\Theta}\|_1\right) = \begin{cases}\lambda, & \theta > 0\\ -\lambda, & \theta < 0\end{cases} \tag{3.7}$$

$$\frac{\partial}{\partial\theta}\left(\lambda\|\mathbf{\Theta}\|_2\right) = 2\lambda\theta \tag{3.8}$$

We want to minimize the energy function given by the regularizers, so we want the push on θ to go in the opposite direction with respect to the derivative, hence the L_1 regularizer acts on θ as $-\lambda$ if $\theta > 0$ and λ if $\theta < 0$, while the L_2 regularizer as $-2\lambda\theta$. Looking at this regularizing actions, we can appreciate how indeed the description made above is true, although not very rigorous.
In general, p-norms are a good choice for regularizers, since thery are always convex. We have seen that ℓ_Θ is convex, and the sum of two convex functions is still convex. This is very important since it means that we can hope to find a closed-form expression for the global optimum to the minimization problem

$$\underset{\Theta}{\operatorname{argmin}}\, \ell_\Theta + \lambda\|\Theta\|_p^2. \tag{3.9}$$

Moreover, note that any p-norm will not be linear in $\mathbf{\Theta}$ because there is at least the absolute value.
Other regularizers induce other desired properties on the parameters. In general, regularization allows us to impose some expected behavior from our learning model, it allows to control the complexity of the model and by controlling the complexity it actually allows us to reduce the need for lots of data because this is imposing some kind of behavior. Note that regularizers are not always defined as penalties included in the loss functions.

Definition 3.2 (Regularization)**.** Any modification that is intended to reduce the generalization error but not the training error.

Other forms include the choice of a representation, *early stopping* and *dropout*.

3.4 Classification

So far we have focused on one of the possible tasks that deep learning can be used for: regression, i.e. learning a mapping from datapoints in $\mathbb{R}^n$ to other points in $\mathbb{R}^n$, usually simply $\mathbb{R}$, so predicting a value given some *features*.
Classification is another very common deep learning task, in which instead we want to predict a *category*, or a *class* for a given datapoint. For example, given some features describing an email, we could be interested to *classify* it as either spam or non-spam; this is an example of *binary classification*.
One could think that classification is just a special case of regression, since we can represent the set of possible categories as natural numbers, so a subset of $\mathbb{R}$. So the idea would be to use the usual regression and then post-process the output value of the regression in some way. In the previous example of spam detection, by applying a threshold to convert it to a binary output.
This is not ideal, since we lose some optimality guarantees: while we know that the optimum value returned by linear regression is an optimum, if we do post-processing and convert it to a categorical output the result is not necessarily an optimum anymore.
What we would like to do instead is to modify our loss to more closely reflect our objective, i.e. to minimize directly the error that the model does when predicting the categorical values. As we will see, linearity shows up again.

Logistic regression *Logistic regression* is one of the simplest approach to classification.
Let $\{x_i, y_i\}$ be the training data, with x_i being some point in $\mathbb{R}^n$ encoding the features of the i-th sample, and y_i being the *label* of the sample, i.e. an encoding of its class. In a binary setting this would mean being a positive or negative sample with respect to some criterion, e.g. being a spam email.

$$y_i = \begin{cases} 0 & x_i \text{ is a negative sample} \\ 1 & x_i \text{ is a positive sample} \end{cases} \tag{3.10}$$

In this setting, we define a new model

$$\hat{f}(x_i) = \underbrace{ax_i + b}_{\text{linear}} \tag{3.11}$$

that makes class *predictions* as follows

$$\hat{y}_i = \begin{cases} 0 & \sigma(\hat{f}(x_i)) \leq 0.5 \\ 1 & \sigma(\hat{f}(x_i)) > 0.5 \end{cases} \tag{3.12}$$

and a new loss

$$\ell_{\boldsymbol{\Theta}}(\{x_i, y_i\}) = \sum_{i=1}^{n} \left(y_i - \sigma(ax_i + b)\right)^2. \tag{3.13}$$

Notice that both the model and the loss are very similar to the regression setting, with the difference that we introduced thresholding and $\sigma(\cdot)$, that is a function called *logistic sigmoid*, defined as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3.14}$$

The logistic sigmoid squashes the real line to within the values 0 and 1 (this is called *saturation* effect) as we can see below:

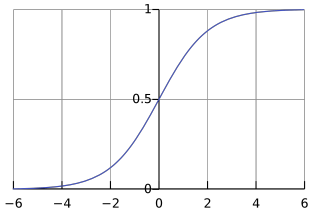


Figure 3.5: Logistic sigmoid.

We like this property of the sigmoid, since as we have seen y_i will take values in $\{0, 1\}$, and therefore we can compare numerically the labels and the model output: the loss is close to zero over samples for which the label is 0 and the model correctly outputs values close to 0 but also samples for which the label is 1 and the model correctly outputs values close to 1.
There is a drawback however: notice that the sigmoid is clearly nonlinear, and even worse non-convex, thus making the whole loss function non-convex. We cannot just take the gradient, set it to 0 and solve for a and b: we don't have any optimality guarantee. Therefore, we can either try to minimize a non convex function or we can try to come up with a different loss.
The modified loss is as follows:

$$\ell_{\boldsymbol{\Theta}}(\{x_i, y_i\}) = \sum_{i=1}^{n} c(x_i, y_i) \tag{3.15}$$

where

$$c(x_i, y_i) = \begin{cases} -ln(\sigma(ax_i + b)), & y_i = 1 \\ -ln(1 - \sigma(ax_i + b)), & y_i = 0 \end{cases} \tag{3.16}$$

is convex. To prove it, we should either prove that the Jensen's inequality holds, or directly show that its second derivative is always non-negative (the actual definition of convex function). In this case we have

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[-ln(\sigma(x))\right] = \frac{\mathrm{d}}{\mathrm{d}x}\left[-ln\left(\frac{1}{1+e^{-x}}\right)\right] = \frac{\mathrm{d}}{\mathrm{d}x}\left[-(ln(1) - ln(1+e^{-x}))\right] = \frac{\mathrm{d}}{\mathrm{d}x}\left[ln(1+e^{-x})\right] \tag{3.17}$$

$$= -\frac{e^{-x}}{1+e^{-x}} = -\frac{-1+1+e^{-x}}{1+e^{-x}} = -1 + \frac{1}{1+e^{-x}} = -1 + \left(1+e^{-x}\right)^{-1} \tag{3.18}$$

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\left[-ln(\sigma(x))\right] = \frac{\mathrm{d}}{\mathrm{d}x}\left[-1 + \left(1+e^{-x}\right)^{-1}\right] \tag{3.19}$$

$$= -(1+e^{-x})^{-2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2} \geq 0 \tag{3.20}$$

so the function is indeed convex. In a similar fashion we could prove that also $-ln(1-\sigma(x))$ is convex so overall $c(x_i, y_i)$ is convex.
Bringing all together we have

$$c(x_i, y_i) = -y_i ln(\sigma(ax_i + b)) - (1 - y_i) ln(1 - \sigma(ax_i + b)). \tag{3.21}$$

since the first term will be present only when $y_i = 1$ and the second only when $y_i = 0$, as per the definition of $c(x_i, y_i)$, and thus we have the following expression for the loss:

$$\ell_{\boldsymbol{\Theta}}(\{x_i, y_i\}) = -\sum_{i=1}^{n} \left(y_i ln(\sigma(ax_i + b)) + (1 - y_i) ln(1 - \sigma(ax_i + b))\right). \tag{3.22}$$

To find a solution we compute the gradient and set it to zero, therefore we obtain:

$$\nabla_\Theta \sum_{i=1}^{n} \left(y_i ln(\sigma(ax_i + b)) + (1 - y_i) ln(1 - \sigma(ax_i + b))\right) = 0 \tag{3.23}$$

$$\sum_{i=1}^{n} \nabla_\Theta \left(y_i ln(\sigma(ax_i + b)) + (1 - y_i) ln(1 - \sigma(ax_i + b))\right) = 0 \tag{3.24}$$

$$\sum_{i=1}^{n} \left(\nabla_\Theta y_i ln(\sigma(ax_i + b)) + \nabla_\Theta (1 - y_i) ln(1 - \sigma(ax_i + b))\right) = 0 \tag{3.25}$$

$$\sum_{i=1}^{n} \left(y_i \nabla_\Theta ln(\sigma(ax_i + b)) + (1 - y_i) \nabla_\Theta ln(1 - \sigma(ax_i + b))\right) = 0 \tag{3.26}$$

In the following, we are going to need the derivative of the sigmoid, which we now derive:

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial}{\partial x} \frac{1}{1+e^{-x}} = \frac{e^{-x}}{\left(1+e^{-x}\right)^2} = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \tag{3.27}$$

$$= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x}) - 1}{1+e^{-x}} = \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \tag{3.28}$$

$$= \sigma(x) \cdot (1 - \sigma(x)) \tag{3.29}$$

Consider the first gradient

$$\nabla_\Theta \underbrace{ln(\sigma(ax_i + b))}_{f(g(h(\Theta)))} \tag{3.30}$$

to compute the derivative of a composed function we can apply the *chain rule* to each partial derivative

$$\frac{\partial}{\partial a} f(g(h(a,b))) = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial a} \tag{3.31}$$

replacing it on our fist gradient we have that

$$\begin{aligned} \frac{\partial}{\partial a} f(g(h(a,b))) &= \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial}{\partial a}(ax_i + b) && (3.32)\\ &= \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot x_i && (3.33)\\ &= \frac{\partial f}{\partial g} \cdot \frac{\partial}{\partial (ax_i + b)} \sigma(ax_i + b) \cdot x_i && (3.34) \end{aligned}$$

Now, recalling the derivative of the sigmoid eq. (3.29)

$$\begin{aligned} \frac{\partial}{\partial a} f(g(h(a,b))) &= \frac{\partial ln(\sigma(ax_i + b))}{\partial \sigma(ax_i + b)} \cdot \sigma(ax_i + b) \cdot (1 - \sigma(ax_i + b)) \cdot x_i && (3.35)\\ &= \frac{1}{\sigma(ax_i + b)} \cdot \sigma(ax_i + b) \cdot (1 - \sigma(ax_i + b)) \cdot x_i && (3.36)\\ &= (1 - \sigma(ax_i + b)) \cdot x_i && (3.37) \end{aligned}$$

and we should continue with the other parameters and considering all terms of the previous expression, but we immediately see that the parameters enter the gradient in a *nonlinear* way since we have the logistic sigmoid which is not a linear transformation. In particular our gradient $\nabla_\Theta \ell_\Theta$ is a *transcendental* equation (*i.e.* some transcendental function[1] appears in the expression, in our case the exponential inside the sigmoid is the transcendental function) and can be proven that for this kind of equations no analytical solution can be found. So to find a solution we are going to do *nonlinear* optimization.

[1] If the function is not a finite series we call it a transcendental function

Chapter 4

Stochastic gradient descent

When dealing with regression, we have seen that not all data distributions are of linear nature, very few actually, and therefore this nonlinearity must be represented by our models.
More in general, it is very frequent having to deal with a loss function with a gradient in which the model parameters enter in a nonlinear way, like we have seen for logistic regression. We have seen how we have optimality guarantees for a closed-form solution (obtained by setting the gradient to $\mathbf{0}$) only for convex functions. Although a nonlinear function is not necessarily a non-convex function (for instance quadrics are not), most are. What this means is that we have no closed-form solutions available, but instead must resort to *nonlinear optimization*.

4.1 Introduction

Nonlinear optimization is a very broad research area, of which we will only consider the branch that is concerned with *first-order methods*, *i.e.* methods that to optimize a function involve the first-order derivatives (in our case it is the gradient) of that function. On the other hand, a second-order method would also involve the use of the second-order derivatives of the function. The most basic of these optimization is known as *Gradient descent*.
Gradient descent (GD) is a first-order *iterative* minimization algorithm. It is iterative since it involves a series of iterated computations to find the *local* minimum of a function, given some *initial conditions*. Notice the word local, meaning that we effectively lose any *global* optimality guarantee when entering the domain of non-convex functions. These functions have multiple local minima, and GD will *converge* to one of these points; which one exactly depends on the initial conditions.
The intuition behind GD is pretty straight-forward: starting from some point in the domain of the function, computing the gradient of the function at that point will tell us the direction of steepest increase (or ascent) of the function itself. Since we want to minimize the function, we take a "step" in the opposite direction, the direction of steepest *decrease* (or descent, hence the name of the algorithm), moving to a new point. In this new point, we repeat the process, and we do so for a certain number of steps, or until some termination condition is met (often referred to as *convergence criterion*).
Ideally, the convergence criterion would be having a perfectly null gradient, meaning that we cannot move (in a linear way) "downwards" anymore; we have reached a stationary point, and

we have effectively reached a (local) minimum of the function[1]. In practice for numerical precision issues this means setting a tolerance threshold under which we can say with confidence that we are in a stationary point.

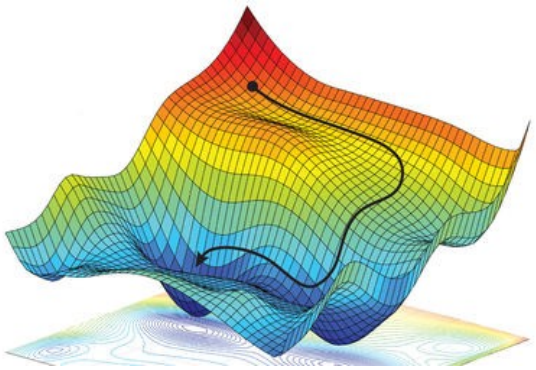


Figure 4.1: Plot of a function from $\mathbb{R}^2$ to $\mathbb{R}$.

In model fitting, the function we want to minimize is the loss function, that has the model parameter space as domain and the real axis as codomain. Therefore the initial point is some initial values for the model parameters; this may be chosen randomly or with some "clever" initialization procedure. We then compute the value of the loss and therefore also the gradient of the loss at that point, which will tell us the (opposite of the) direction of steepest descent. So, we now have the direction in the parameter space that will (locally) make the loss decrease the most. How much should we "move" (i.e. change the parameters) in that direction? This is the *step size*, that can be thought as a *hyperparameter*.
Having chosen a suitable step size we can make our step in that direction, moving to a new point that will be a new set of values for the model parameters. From this point in the parameter space we will compute again the loss function and iterate the process until the convergence criterion is met, tracing a sort of path (see fig. 4.1) that ends in a local minimum.
Let's outline this procedure more formally:

1. Define an initial condition for the algorithm, *i.e.* a point $\Theta^{(0)} \in \mathbb{R}^n$;
2. Iteratively update the parameters with the following update law:

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \alpha \nabla \ell_{\Theta^{(t)}}; \tag{4.1}$$

3. Stop iterating when a convergence criterion is satisfied, *e.g.*

$$\|\boldsymbol{\nabla} \ell_{\Theta^{(t)}}\| \leq \epsilon$$

for a suitable small positive value ϵ.

[1] or a *saddle point*, if we are *very* unlucky; in practice this is never a problem, since theoretically they become exponentially more rare the higher it is the dimension of the domain, and also practically being *exactly* in a saddle point is almost impossible given the finite numerical precision of machines

4.2 Gradient Properties

4.2.1 Orthogonality and steepest ascent

We have not yet provided a formal justification of the claim that the gradient of a function at a point, which is the vector of the partial derivatives, is in direction of the steepest increase of the function at that point.
We therefore briefly introduce the concept of *directional derivative*, an extension of the "usual" derivatives in the simple one-dimensional domain $\mathbb{R}$. While on the real axis there is only one direction, in $\mathbb{R}^n$, starting from $n = 2$ upwards, there is no fixed direction to evaluate the derivative of a function.
The directional derivative $\frac{\mathrm{d}f}{\mathrm{d}\mathbf{v}}$ generalizes the concept of partial derivative $\frac{\partial f}{\partial x}$, which assumes that the direction in which we take the derivative is one where only one of the variables can change, while the others are fixed (one of the canonical axes). This assumption is lost when taking the derivative in a general direction $\mathbf{v}$. In practice, this means that (potentially) all the variables of the function change in that direction according to some law. For example, suppose we have the polynomial function

$$f : \mathbb{R}^2 \to \mathbb{R} \quad f(x, y) = x^3 + y^2$$

and we want to take the directional derivative

$$\frac{\mathrm{d}f}{\mathrm{d}\mathbf{v}} \quad \text{with } \mathbf{v} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}^\top$$

then the independent variables x, y are not independent anymore, but are bound to the line of slope $\frac{y}{x} = \frac{\sqrt{2}}{2}\frac{2}{\sqrt{2}} = 1$ (note that to be a "pure" direction, the vector must be a unit vector), therefore we can parametrize them with a new independent variable t such that $x(t) = y(t) = \frac{\sqrt{2}}{2}t$. Now taking the directional derivative boils down to a substitution and a simple derivative in $\mathbb{R}$:

$$\frac{\mathrm{d}f}{\mathrm{d}\mathbf{v}} = \frac{\mathrm{d}}{\mathrm{d}f}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\sqrt{2}}{2}(t^3 + t^2) = \frac{\sqrt{2}}{2}(3t^2 + 2t) = \frac{\sqrt{2}}{2}(3x^2 + 2y).$$

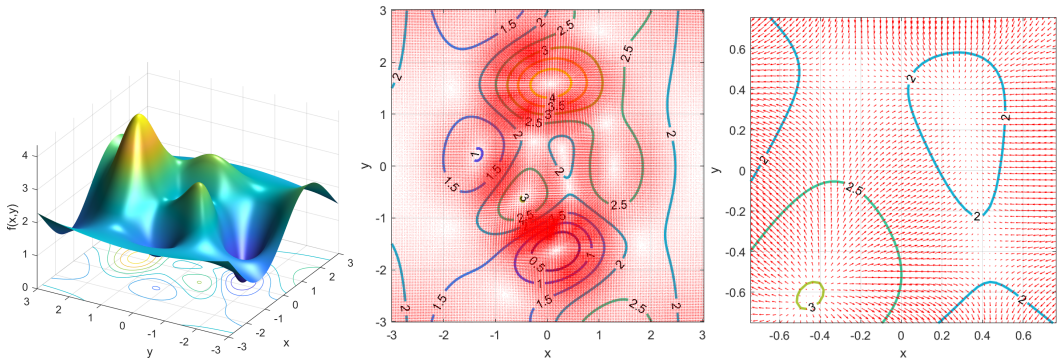


Figure 4.2: Computing the gradient of the function at "every" point of the domain leads to a *vector field*, that visualizes the "flow" of the function, from points of lower values to points of higher values. We seek to follow the opposite flow. Indeed, in the figures the negative gradient is plotted, with regions with higher density having points with larger gradients in magnitude.

How does the notion of directional derivative lead us to the gradient being in the direction of steepest increase?

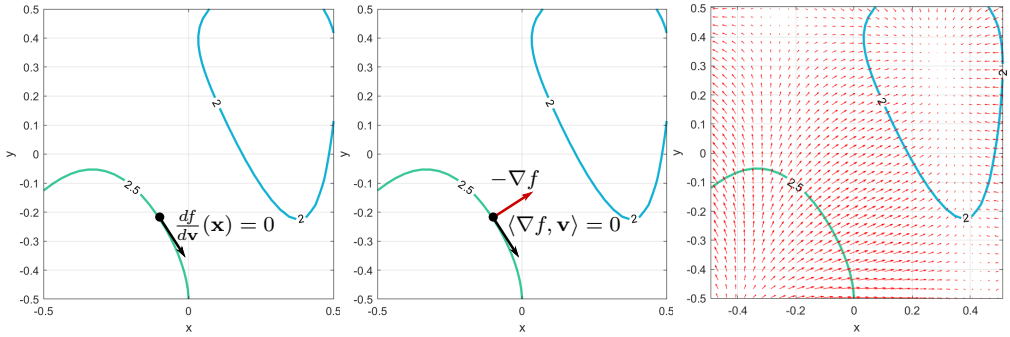


Figure 4.3: Relation between isocurves, directional derivatives, gradient, and function increase.

The *isocurves* of a function are the the curves (or hyper-surfaces) on which the value of a function does not change. This means that taking the directional derivative of the function, in a point on the curve and along a direction $\mathbf{v}$ tangent to the curve, this directional derivative will be zero, since locally the function is not changing in that direction (although it will, globally, since this is only a local linear approximation of the behavior of the function). Then, it can be shown that $\langle \boldsymbol{\nabla} f, \mathbf{v} \rangle = 0$, which means that the gradient is orthogonal to the level curve, and it can be further shown that it is oriented towards level curves of higher values (instead of lower values). This is summarized in fig. 4.3.

4.2.2 Differentiability

We have seen how the gradient is central to the gradient descent algorithm, but do all (loss) functions behave so nicely, or is the gradient something that we cannot take for granted?
Just like with functions $f : \mathbb{R} \to \mathbb{R}$, not all functions have well-defined derivatives across all their domain, *i.e.* they are *not differentiable*. Gradient descent requires the function under consideration to be differentiable, otherwise the gradient may not be well defined in certain points and the algorithm may fail abruptly. However, in $\mathbb{R}^n$ things are trickier than in $\mathbb{R}$, to establish the differentiability of a function. Indeed, if we already know that the function has a *continuous* gradient, this means that the function is differentiable (sufficient condition), but we would rather like an "inverse" sufficient condition. It turns out that even if a function f has all the partial derivatives (recall, they are the elements of the gradient vector $\boldsymbol{\nabla} f = \left(\frac{\partial f}{\partial x_1} \quad \dots \quad \frac{\partial f}{\partial x_n} \right)$) well defined, this is only a necessary (not sufficient) condition for differentiability.
Suppose we have the following function:

$$f(x, y) = \begin{cases} 0 & \text{at } (0, 0) \\ \frac{x^2 y}{x^2 + y^2} & \text{elsewhere} \end{cases}$$

The partial derivative $\frac{\partial}{\partial y} f(x, y)$ is discontinuous at $(0, 0)$, since it is

$$\frac{\partial}{\partial y} f(x, y) = \frac{x^2 (x^2 - y^2)}{(x^2 + y^2)^2}$$

so

$\frac{\partial}{\partial y} f(x,0) = \frac{x^4}{x^4} = 1$ for $x \neq 0$

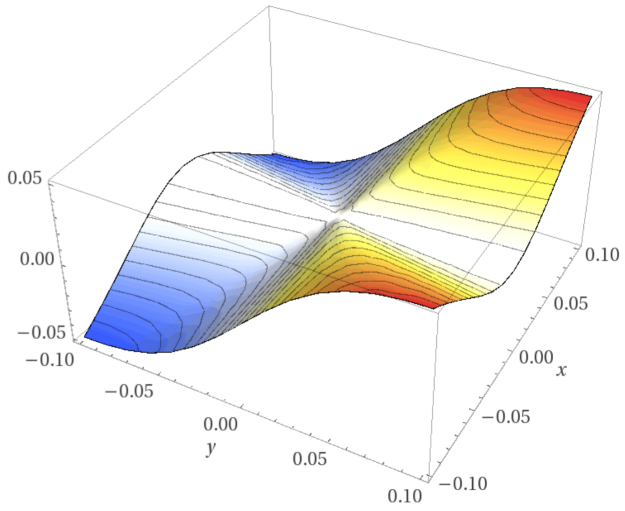
$\frac{\partial}{\partial y} f(0,y) = \frac{0}{y^4} = 0$ for $y \neq 0$

$\Rightarrow \frac{\partial}{\partial y} f(x,y)$ discontinuous at $(0,0)$

the partial derivatives of f are *defined everywhere*, but are *discontinuous* at the origin

$\Rightarrow f$ is *not differentiable*

.



One does not usually encounter such loss functions, but it is not rare to come across non-differentiable loss functions. In these situations one should re-engineer the loss function in some way (e.g. the *reparametrization trick* for VAEs, as we will see) to make it differentiable.

4.2.3 Stationary points

A stationary point, for a differentiable function, is a point in which all the partial derivatives are zero, *i.e.* the gradient is null. From the perspective of gradient descent this means that the method "gets stuck" in these stationary points:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \alpha \cancelto{0}{\nabla f(\mathbf{x}^{(t)})}. \tag{4.2}$$

Why is this a problem? Are we not seeking these points, since they are the minima of the functions? Well, (local) minima are one kind of stationary points, that also include local maxima and saddle points, points in which the function increases along an axis and decreases along another axis. To understand which is which analytically, one resorts to the second (partial) derivative test, which we will not delve into. Going back to the gradient descent, the type of stationary point to which the algorithm converges depends on the initialization; for instance, if we start precisely in a local maxima or a saddle point (although very unlikely in practice), the algorithm will immediately stop, since the gradient is zero, even though those are *not* minima of the function.

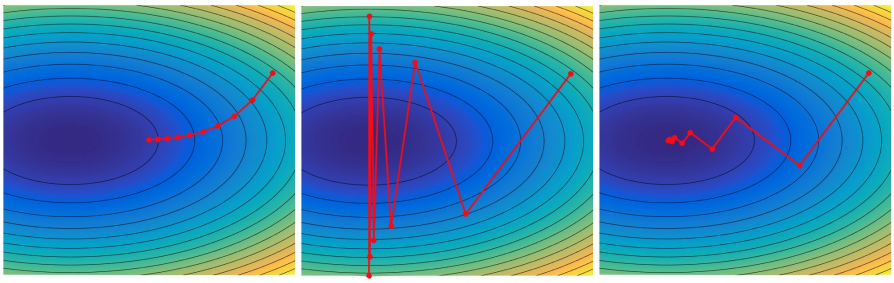
4.3 Learning Rate

In the update law in eq. (4.1), we have not yet clarified the role of α. This is a *hyperparameter*, meaning a parameter that does not belong to the model and does not influence the model directly, but does so indirectly by affecting the way it is fitted to the data. This parameter is always positive (otherwise we would maximize the loss function, by moving in the direction of the gradient) and is called *learning rate*.
The name is due to the fact that it influences the length of a "learning" step of the gradient descent algorithm, and hence the rate at which the algorithm makes progress. It is not precisely

the step length itself, since that would be

$$\left\|\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}\right\| = \left\|\alpha\nabla f(\mathbf{x}^{(t)})\right\| = \alpha\left\|\nabla f(\mathbf{x}^{(t)})\right\|. \tag{4.3}$$

Nonetheless, the direct proportionality that stands means that the value of this parameter can heavily influence the evolution of the algorithm. If this value is too small, then we have slow convergence speed, *i.e.* we will need more steps to reach a minimum. If it is too large, we risk *overshooting*, *i.e.* the algorithm starts oscillating since even though the minimum is indeed in the direction of the gradient, reaching it would require a smaller step and therefore the algorithm keeps going back and forth around it, something that is often referred to as *divergence* (or *convergence issues*).



small α | large α | optimal α $\arg\min_\alpha f(\mathbf{x}^{(t)} - \alpha\nabla f(\mathbf{x}^{(t)}))$

Figure 4.4: Comparison of values of the learning rate α.

Since we move in straight lines, and the function is nonlinear, we cannot know what is the overall best value for α; if this was possible, we could just make one step with the optimal α, i.e. a single computation, a closed-form solution, but as we know this is not (guaranteed to be) possible in the nonlinear setting. Indeed, recall that the gradient is only a local linear approximation of the behavior of the function, so we cannot hope to have just one step length that, when executed along a straight line, works well everywhere.
Instead, we can locally adjust the value of α according to the local information we have at a certain point, with a technique called *line search*. What it does is very simple: once the direction of steepest descent is known ($-\nabla f(\mathbf{x}^{(t)})$), we seek the value α such that taking a step of size $\alpha\left\|\nabla f(\mathbf{x}^{(t)})\right\|$ in that direction makes the function decrease the most. So we have an optimization procedure inside an optimization algorithm.

4.3.1 Decay

A line search is not the only way through which we can define a value for α. In fact, the learning rate can be *adaptive* or follow a *schedule*, so having it update each learning step along with the parameters.
For instance, one could have one of the following schedules

- $\alpha^{(t+1)} = \left(1 - \frac{t}{\rho}\right)\alpha^{(0)} + \frac{t}{\rho}\alpha^{(\rho)}$,
- $\alpha^{(t+1)} = \frac{\alpha^{(t)}}{1+\rho t}$, or
- $\alpha^{(t+1)} = \alpha^{(0)}e^{-\rho t}$

in which ρ is a *decay* parameter. This is motivated by the idea that initially we need large steps to swiftly progress in the general direction of the minimum. Later on, once we get closer and closer to the point, we need smaller steps to reach convergence. Of course, the decay parameter can enter the update law for α in many ways: balancing a linear interpolation over time between an initial value $\alpha^{(0)}$ and a final value $\alpha^{(\rho)}$, or monotonically decreasing α over time, in a linear way or even an exponential way. Needless to say, there is no "best recipe" here, one approach may (or may not) work better than another depending on the specific setting under consideration.

4.3.2 Momentum

Another approach takes inspiration from physics, since after all we have used the analogy of "moving" in the parameter space so far. Moving involves velocity, and bodies with mass possess *momentum*, that is conserved over time (by the principle of conservation of momentum) and keeps them in motion, unless an external force (like friction, taking away momentum, or a force, causing an acceleration effect and hence adding more momentum) steps in to modify it. If we imagine our moving estimate of the best parameters for the model during gradient descent to be a point particle, with unitary mass, its momentum $\mathbf{p} = m\mathbf{v}$ coincides with its velocity $\mathbf{v}$. In the case of simple gradient descent we have

$$\begin{aligned} \mathbf{v}^{(t+1)} &= -\alpha \nabla f(\mathbf{x}^{(t)}) \\ \mathbf{x}^{(t+1)} &= \mathbf{x}^{(t)} + \mathbf{v}^{(t+1)} \end{aligned} \tag{4.4}$$

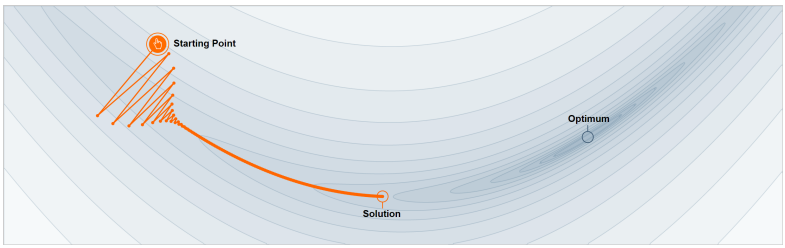
so each step the velocity is simply given by the "acceleration" that the gradient imposes on the body at that point. We can imagine the surface made from the function graph as a landscape, made of peaks, valleys and pitfalls, and the vector field the gradient of this function induces on the parameter space as the gravitational field. Therefore, at each point the gradient imposes a gravitational force, that for a unitary mass body immediately translates to acceleration. So our parameter estimate $\boldsymbol{\Theta}^{(t)}$, that we want to move as low as possible, towards the (hopefully global, but more likely local) minimum of the loss function, is rolling downwards in this landscape under the effect of the loss function gradient.
However, each step the body discards the velocity imposed by the previous gradient, and takes as new velocity the new gradient at the new point. Phisically, past accelerations are accumulated since the velocity does not reset back to zero at each "next step", like it is happening here. Therefore, we add in our conservation of momentum idea, to have an effect of *accumulating* past gradients, but with gradients computed in previous steps having less and less importance, much like friction (with coefficient λ) takes away momentum for a body in motion on a surface.

$$\begin{aligned} \mathbf{v}^{(t+1)} &= \lambda \mathbf{v}^{(t)} - \alpha \nabla f(\mathbf{x}^{(t)}) \quad \textit{momentum} \\ \mathbf{x}^{(t+1)} &= \mathbf{x}^{(t)} + \mathbf{v}^{(t+1)} \end{aligned} \tag{4.5}$$

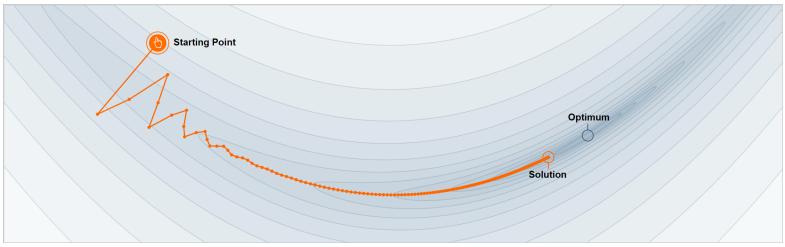
Previously, the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a sequence of gradients are. Also, the larger λ (with $0 \leq \lambda < 1$) is relative to α, the more previous gradients affect the current direction. The step size is largest when many successive gradients point in exactly the same direction. If the momentum algorithm always observes the same gradient $\boldsymbol{\nabla} f$, then it will accelerate in the direction of $-\boldsymbol{\nabla} f$, until reaching a terminal velocity (like a body in free fall motion reaching terminal velocity due to the drag imposed by air resistance) where the size of each step is

$$\frac{1}{1-\lambda} \alpha \|\nabla f\|. \tag{4.6}$$

It is thus helpful to think of the momentum hyperparameter λ in terms of $\frac{1}{1-\lambda}$. For example, $\lambda = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the standard gradient descent algorithm.
But why exactly should momentum help the algorithm? In the fig. 4.5, we can see that with large enough λ, we have an acceleration effect that helps the convergence speed. Furthermore, keeping the analogy of a body rolling downwards under the effect of gravity, if we imagine the pitfalls to be local minima, since we are actually interested in global minima, we would like for the body to not get stuck in pitfalls that are not the lowest ones. Picture the situation: the body ends in a pitfall, but it has so much momentum left that it is able to escape it and (hopefully) keep rolling downwards to end up in a lower pitfall.



(a) Without momentum



(b) With momentum

Figure 4.5: Acceleration effect of momentum in situations with small step size.

Let's try to be more formal now. The update rule of standard gradient descent, once unrolled, has the following form:

$$\begin{aligned}
\mathbf{x}^{(1)} &= \mathbf{x}^{(0)} - \alpha \nabla f(\mathbf{x}^{(0)}) \\
\mathbf{x}^{(2)} &= \mathbf{x}^{(1)} - \alpha \nabla f(\mathbf{x}^{(1)}) \\
&= \mathbf{x}^{(0)} - \alpha \nabla f(\mathbf{x}^{(0)}) - \alpha \nabla f(\mathbf{x}^{(1)}) \\
&\vdots \\
\mathbf{x}^{(t+1)} &= \mathbf{x}^{(0)} - \alpha \sum_{i=1}^{t} \nabla f(\mathbf{x}^{(i)}).
\end{aligned} \tag{4.7}$$

Repeating the same procedure with the momentum we get

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(0)} + \alpha \sum_{i=1}^{t} \frac{1 - \lambda^{t+1-i}}{1 - \lambda} \nabla f(\mathbf{x}^{(i)}) \tag{4.8}$$

in which we can explicitly see the effect of accumulating gradients. Each gradient is multiplied

by a factor that is exponentially closer to zero the furthest back in the past the gradient is, since $\lambda \leq 1$.
This generalizes to the following form

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(0)} + \alpha \sum_{i=1}^{t} \Gamma_i^t \nabla f(\mathbf{x}^{(i)}) \quad \text{for some diagonal matrix } \Gamma_i \tag{4.9}$$

and many optimization algorithms (ADAM, AdaGrad, etc.) derived by gradient descent, take this form, with different specific rules for the diagonal matrix, that gives more or less importance to the various components of the gradient vector in a linear and decoupled way.

4.4 Gradient Descent for Deep Learning: Stochastic Gradient Descent

So far we have been very focused on the properties of gradient descent, more from an optimization perspective than from a deep learning one. Indeed, if we think about it, we don't really seek the global minimum of a loss function, because that would mean a perfect fit to the data, *i.e.* overfitting and loss of generalization power of the model. So, even though gradient descent brings forth no optimality guarantee, as deep learning practitioners this is not trouble for us, and we are happy to apply it to nonconvex problems, like optimization of a multi-layer perceptron that we will see later. Actually gradient descent is often applied to solve even convex problems, like the optimization of the logistic regression, from which we started, having no closed form solution, or even linear regression, for which we do have a closed form solution. Recall however that this solution involves the inversion of a (potentially very big) matrix, something that is not efficient (while on the other hand gradient descent is an iterative algorithm having low cost for each step) and can be affected by numerical stability / bad conditioning issues.
To recap, in the deep learning setting the gradient descent algorithm is all about the following update rule:

$$\boldsymbol{\Theta}^{(t+1)} \leftarrow \boldsymbol{\Theta}^{(t)} - \alpha \boldsymbol{\nabla}_{\boldsymbol{\Theta}^{(t)}} \ell(\mathbf{y}, \mathbf{f}_{\boldsymbol{\Theta}}(\mathbf{X})) \tag{4.10}$$

where $\boldsymbol{\Theta}$ is the vector of all the model parameters, α is the learning rate, ℓ is the loss function, evaluated on the true output vector $\mathbf{y}$ and the predicted output vector

$$\mathbf{f}_{\boldsymbol{\Theta}}(\mathbf{X}) = \begin{pmatrix} \mathbf{f}_{\boldsymbol{\Theta}}(\mathbf{x}_1) & \dots & \mathbf{f}_{\boldsymbol{\Theta}}(\mathbf{x}_n) \end{pmatrix}^\top.$$

So, each parameter gets updated so as to decrease the loss:

$$\boldsymbol{\Theta}_i \leftarrow \boldsymbol{\Theta}_i - \alpha \frac{\partial \ell}{\partial \boldsymbol{\Theta}_i}. \tag{4.11}$$

However, in this setting we have to deal with practical problems that the well defined mathematical solutions do not care for. Some of these are:

- The model might have millions of parameters, hence making the update rule computationally intensive;
- The loss function ℓ can be non-convex and also non-differentiable in some cases, so in principle one could not even begin to apply gradient descent;
- Even if all is set correctly, one might encur in numerical instability issues, leading to phenomena like exploding / vanishing gradients.

In fact, gradient descent in its standard form is actually *impractical*. Recall that the loss is usually defined over n training examples:

$$\ell_\Theta(\{x_i, y_i\}) = \frac{1}{n}\sum_{i=1}^{n} \hat{\ell}_\Theta(\{x_i, y_i\}) \tag{4.12}$$

in which for instance $\hat{\ell}$ may be a term of the mean squared error loss function $(y_i - f_\Theta(x_i))^2$. This requires computing the gradient for each term in the summation:

$$\nabla\ell_\Theta(\{x_i, y_i\}) = \frac{1}{n}\sum_{i=1}^{n} \nabla\hat{\ell}_\Theta(\{x_i, y_i\}). \tag{4.13}$$

Two bottlenecks make gradient descent impractical: the number of examples n and the number d of parameters, *i.e.* the size of the parameter vector $\mathbf{\Theta}$.
We can attempt to reduce one of the two bottlenecks: the number of examples. Of course, data is precious to us, so we do not want to simply discard data. Instead, we "sacrifice the gradient" of the loss function, meaning we do not compute an exact gradient but rather an estimate of it, introducing the concept of *batch*.
Suppose that we have a training set $\mathcal{T} = \{\mathbf{x}_i, y_i\}$, then the exact gradient of the loss function would be

$$\nabla\ell_\Theta(\mathcal{T}) = \frac{1}{n}\sum_{i=1}^{n} \nabla\hat{\ell}_\Theta(\mathcal{T}).$$

Instead, we compute $\nabla\ell_{\mathbf{\Theta}}$ only for a small representative subset of $m \ll n$ examples, that we call a mini-batch $\mathcal{B}$:

$$\frac{1}{m}\sum_{i=1}^{m} \nabla\hat{\ell}_\Theta(\mathcal{B}) \approx \frac{1}{n}\sum_{i=1}^{n} \nabla\hat{\ell}_\Theta(\mathcal{T}) \tag{4.14}$$

The mini-batch $\mathcal{B} \subset \mathcal{T}$ is a subset of the training data drawn uniformly, so at each learning step the algorithm uses a different minibatch, eventually covering all the training data. The true gradient $\nabla\ell_\Theta$ is approximated, but with a significant *speed-up*.
This is *Stochastic Gradient Descent* (SGD), which is outlined as follows:

1. Initialize $\mathbf{\Theta}$;
2. Pick a mini-batch $\mathcal{B}$;
3. Update with the downhill step (use momentum if desired):

$$\mathbf{\Theta} \leftarrow \mathbf{\Theta} - \alpha\nabla\ell_{\mathbf{\Theta}}(\mathcal{B});$$

4. Go back to step (2).

When steps (2)-(4) cover the entire training set $\mathcal{T}$, we say that the algorithm has performed an *epoch*. Like gradient descent, the algorithm proceeds for many epochs. The update rule has now constant computational cost (as a function of the fixed size m of the mini-batch), regardless of the size of the training set $\mathcal{T}$.
What is the price of this decreased computational cost?

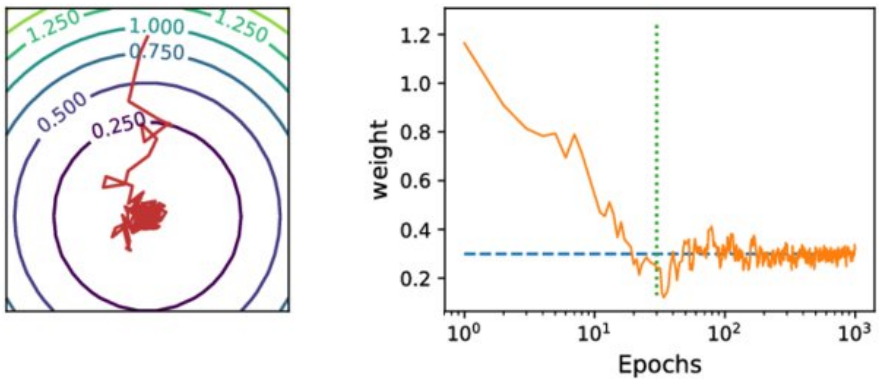


Figure 4.6: Empirical observation of oscillatory behavior at convergence.

Empirically we see that the random sampling that drives the gradient estimation induces oscillations, that make so that SGD does not stop at the minimum. In fact, at each step SGD sees a "different" loss function, one that takes into account the mini-batch only, that does not necessarily have a minimum in that point.
Furthermore, analytically we can see that gradient descent enjoys better convergence rates than stochastic gradient descent. Let's restrict ourselves to convex problems for simplicity. Let

$$\ell(f_\Theta) = \frac{1}{n}\sum_{i=1}^{n} \hat{\ell}(f_\Theta) = \frac{1}{n}\sum_{i=1}^{n} \hat{\ell}(\{f_\Theta(x_i), y_i\})$$

be the loss function of the learning problem, which we approach with a parametric model f_Θ. For these problems there is a true minimizer

$$f^* = \arg\min_f \ell(f) \tag{4.15}$$

and so we consider the inequality

$$|\underbrace{\ell(f_\Theta)}_{\text{GD/SGD}} - \underbrace{\ell(f^*)}_{\text{true}}| < \underbrace{\rho}_{\text{accuracy}} \tag{4.16}$$

which tells us how far off is our model compared to the true minimizer. ρ is an *accuracy* measure that we establish, meaning that once the equality holds, we stop the algorithm since we have reached the desired accuracy. This reasoning produces asymptotic upper bounds on the evolution of the two algorithms:

	cost per iteration	**iterations to reach ρ**
GD	$O(nd)$	$O(\kappa \log \frac{1}{\rho})$
SGD	$O(d)$	$\frac{\nu\kappa^2}{\rho} + o(\frac{1}{\rho})$

where

- n is the number of training examples;
- d is the number of parameters;
- κ, ν are constants related to the conditioning of the problem, we do not need to worry about them, just know that they are constants.

We see that SGD has indeed slower asymptotic convergence, but it has been argued that for machine learning tasks faster convergence presumably corresponds to overfitting, and therefore it is not worthwile to seek convergence faster than $O(\frac{1}{\rho})$. Furthermore, SGD does not depend on the number of examples, implying *better generalization*. Going back to the convergence speed point of view, the asymptotic analysis obscures many advantages that stochastic gradient descent has after a small number of steps. With large datasets, the ability of SGD to make rapid initial progress while evaluating the gradient for only very few examples outweighs its slow asymptotic convergence.
Here we list some final practical considerations about SGD:

- It needs the training data to be *shuffled* to avoid data bias, something that would invalid the approximation, i.e.
$$\frac{1}{m}\sum_{i=1}^{m}\nabla\hat{\ell}_\Theta(\mathcal{B}) \;\not\approx\; \frac{1}{n}\sum_{i=1}^{n}\nabla\hat{\ell}_\Theta(\mathcal{T}).$$
- Each mini-batch can be processed in *parallel*, something that goes well with the increased availability and power of parallel architectures like GPGPU (General Purpose Graphic Processing Unit), more and more used for deep learning tasks. In this case, the batch size is limited by hardware and the size of the specific representation of the data in memory.
- Small mini-batches can offer a *regularizing* effect, by introducing variance in the estimation of the gradients, that prevents the algorithm from reaching the true minimum and thus from overfitting. On the other hand, very small batches introduce too high of a variance (in the limit, online learning, using only one data point at a time), and it is necessary to use a small (even better if decaying) learning rate to maintain stability.

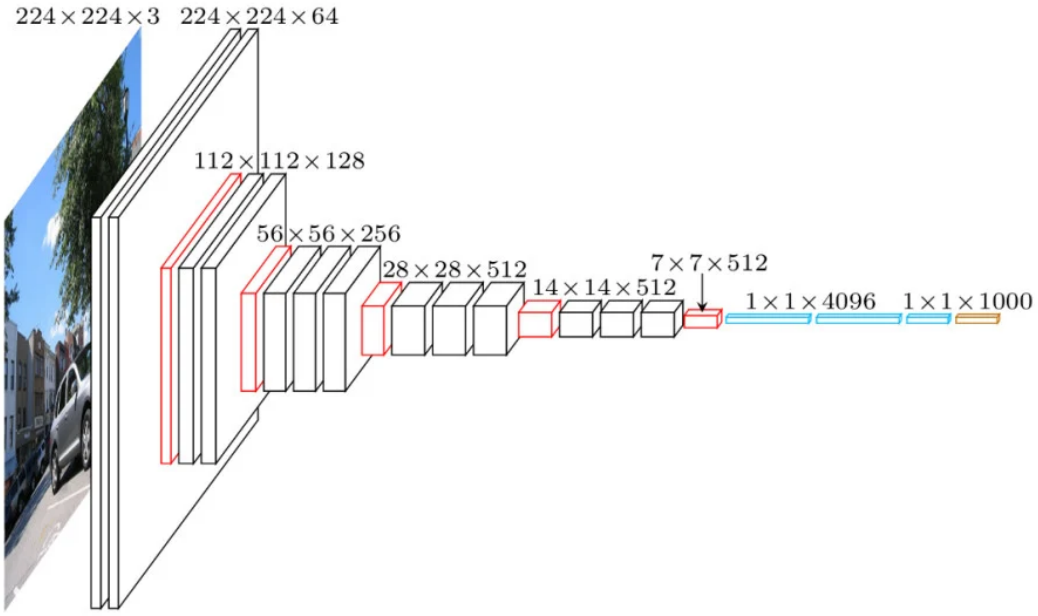
To wrap up all the previous considerations, we can say that SGD can find a *low value* of the loss function quickly enough to be useful, even if it's not a minimum.

Chapter 5

Multi-layer perceptron and back-propagation

5.1 Introduction

In deep learning, we deal with *highly parametrized models* called *deep neural networks*, like the one we can see below.



Each block has a predefined structure (e.g., a *linear map*) and is defined in terms of *unknown parameters* θ. Finding the values for the parameters is called *training*, and this is done by minimizing a function called *loss* function. As we have seen in the previous chapter, usually this process requires nonlinear optimization through an iterative procedure like *stochastic gradient descent*, that requires computing gradients, so we need a way to compute these efficiently.

Moreover, many interesting phenomena are *highly nonlinear*. Our task is to choose a good learning model which is nonlinear enough to capture these phenomena, keeping in mind that a powerful model should be as *universal* as possible, e.g. it should be able to represent the widest possible class of functions.
The general recipe is always the following:

- *Fix* the general form for the parametric model.
- *Optimize* for the parameters.

It is worth noting that the model should also be easy to work with.

5.2 Deep networks

5.2.1 Deep composition

The simplest DNN we may create is the composition of two models:

$$f \circ f(\mathbf{x}) \tag{5.1}$$

This does not seem very "deep", and indeed there is not a strict definition of what makes a network deep or shallow. More notably, if the two functions are linear, also their composition is linear.

$$\underbrace{f}_{\text{linear}} \circ \underbrace{f}_{\text{linear}}(\mathbf{x}) \implies \underbrace{f \circ f}_{\text{linear}}(\mathbf{x}) \tag{5.2}$$

As we will see, what makes DNNs interesting is their theoretical capacity of representing every function, even highly nonlinear ones. This is of course not possible if our DNN is just a linear function, so the least we can do is replace the second function with a nonlinear one:

$$\sigma \circ f(\mathbf{x}) \tag{5.3}$$

σ is sometimes refered to as *non-linearity*, or *activaction function*. If σ is the logistic function, we have the *logistic regression* model.
Now that we have a simple nonlinear model, we can build a more complex nonlinear model by composing multiple nonlinear *layers*. Consider for example multiple *layers* of logistic regression models:

$$\text{output} \leftarrow \underbrace{(\sigma \circ f)}_{\text{layer n}} \circ (\sigma \circ f) \circ \cdots \circ \underbrace{(\sigma \circ f)}_{\text{layer 1}}(\mathbf{x}) \leftarrow \text{input} \tag{5.4}$$

This already defines a *deep model*. Note that function composition is associative, so parentheses are not necessary. Note furthermore that the f and σ in the above expression could be different linear transformations and nonlinearities.
More in general, there are various activation functions, with different properties; the logistic sigmoid for example is continous

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.5}$$

while the *Rectified Linear Unit* (ReLU)

$$\sigma(x) = \max\{0, x\} \tag{5.6}$$

has a discontinous gradient, even if the function itself is continous.

5.2.2 MLP

We call the composition with linear f and nonlinear σ:

$$g_{\boldsymbol{\Theta}}(\mathbf{x}) = (\sigma \circ f_{\boldsymbol{\Theta}_n}) \circ (\sigma \circ f_{\boldsymbol{\Theta}_{n-1}}) \circ \cdots \circ (\sigma \circ f_{\boldsymbol{\Theta}_1})(\mathbf{x}) \tag{5.7}$$

a *multi-layer perceptron* (MLP) or *deep feed-forward neural network*. As can be seen in the expression, the parameters or *weights* of the MLP are scattered across the layers; these only refer to the parameters of the linear transformations, while the nonlinearities are fixed.
Each layer outputs an intermediate *hidden representation*:

$$\mathbf{x}_{\ell+1} = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_\ell) \tag{5.8}$$
$$\mathbf{x}_{\ell+1} = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_\ell + \mathbf{b}_\ell) \tag{5.9}$$

where we encode the weights at layer ℓ in the matrix $\mathbf{W}_\ell$ and bias $\mathbf{b}_\ell$.
Remark: The *bias* can be included inside the weight matrix by writing:

$$\mathbf{W} \mapsto \begin{pmatrix} \mathbf{W} & \mathbf{b} \end{pmatrix}, \quad \mathbf{x} \mapsto \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \tag{5.10}$$

because each f is *linear in the parameters* just like in linear regression (note that the transformation with respect to the input x would be affine, not linear).

Hidden units

At each *hidden layer* ℓ we have:

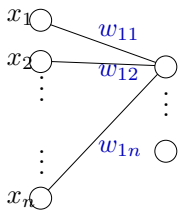
$$\mathbf{x}_{\ell+1} = \sigma_\ell(\mathbf{W}_\ell \mathbf{x}_\ell) \tag{5.11}$$

Each row of the weight matrix is called a *neuron* or *hidden unit*:

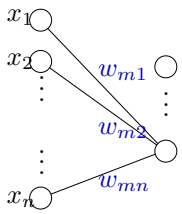
$$\mathbf{W}\mathbf{x} = \begin{pmatrix} \text{— unit —} \\ \vdots \\ \text{— unit —} \end{pmatrix} \begin{pmatrix} | \\ \mathbf{x} \\ | \end{pmatrix}. \tag{5.12}$$

Mathematically, each layer is a vector-to-vector function $\mathbb{R}^p \to \mathbb{R}^q$. Each layer has q units acting *in parallel*, and each unit acts as a scalar function $\mathbb{R}^p \to \mathbb{R}$, computing a certain component of the output vector.

Single layer illustration

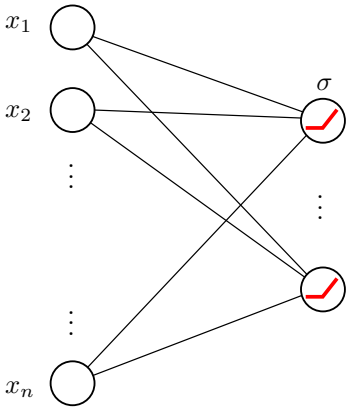


$$\sigma(\mathbf{Wx}) = \sigma \circ \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \cdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \sigma \circ \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \tag{5.13}$$



$$\sigma(\mathbf{Wx}) = \sigma \circ \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \cdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \sigma \circ \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \tag{5.14}$$

To visualize the computation we usually take the hidden input representation $\mathbf{x}$ (possibly the input to the entire network), draw the corresponding n nodes, then do the same for the output, drawing m nodes. Now, for every output node, we see which input nodes intervened to give raise to its value and draw the resulting edge. Each edge has a weight, and this is the corresponding element in the matrix.



For this model, the resulting network will be fully connected, but this will not be always the case.

Output layer

The output layer determines the co-domain of the network:

$$\mathbf{y} = (\sigma \circ f) \circ (\sigma \circ f) \circ \cdots \circ (\sigma \circ f)(\mathbf{x}) \tag{5.15}$$

If σ is the logistic sigmoid, then the entire network will map:

$$\mathbb{R}^p \to (0,1)^q \tag{5.16}$$

For generality, it is common to have a *linear* layer at the output:

$$\mathbf{y} = f \circ (\sigma \circ f) \circ \cdots \circ (\sigma \circ f)(\mathbf{x}) \tag{5.17}$$

mapping:

$$\mathbb{R}^p \to \mathbb{R}^q \tag{5.18}$$

Deep ReLU networks

Adding a linear layer at the output:

$$\begin{aligned} \mathbf{y} &= f \circ (\sigma \circ f) \circ \cdots \circ (\sigma \circ f)(\mathbf{x}) \\ &= f \circ \sigma(\cdots)(\mathbf{x}) \end{aligned} \tag{5.19}$$

expresses $\mathbf{y}$ as a linear combination of "ridge functions" $\sigma(\cdot)$. *ReLU* $\sigma(x) = \max\{0, x\}$ is a piecewise-linear function, as can be seen in fig. 5.1.

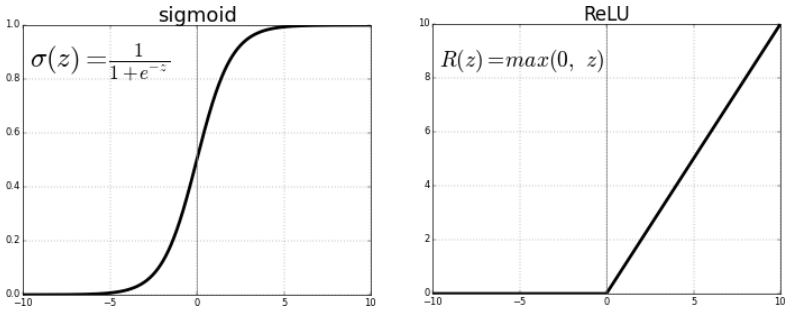


Figure 5.1: Activation functions.

If we linearly combine multiple piecewise-linear function, we get a function which is still piecewise-linear. The result of the composition of linear functions with ReLU as activactions is called *Deep ReLU network*.
For a 2-layer network taking 2-dimensional input and returning a scalar with *ReLU* as activation we can see that the resulting surface is piecewise-linear.

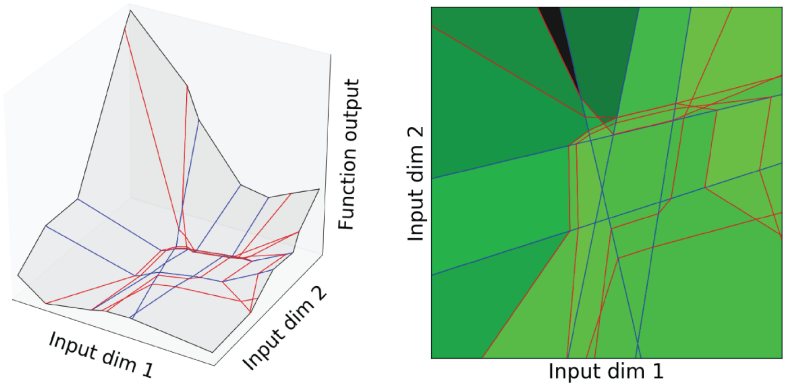


Figure 5.2: Landscape of the function for a 2-layer deep ReLU network. Best viewed in color.

The blue and red edges are produced by the first and second layer.
The class of functions that can be represented with deep ReLU networks is thus the class of piecewise-linear functions.

5.2.3 Universality

What class of functions can we represent with a MLP?
If σ is sigmoidal, we have the *Universal Approximation Theorem* (UAT).

Theorem 5.1 (Universal Approximation Theorem)**.** *For any compact set $\Omega \subset \mathbb{R}^p$, the space spanned by the functions $\phi(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x}+\mathbf{b})$ is dense in the set of continuous functions $\mathcal{C}(\Omega)$ for the uniform convergence. Thus, for any continuous function f and any $\epsilon > 0$, there exists $q \in \mathbb{N}$ and weights $\{u_k \in \mathbb{R}\}_{k=1}^{q}$ such that*

$$|f(\mathbf{x}) - \sum_{k=1}^{q} u_k \phi(\mathbf{x})| \leq \epsilon \qquad \text{for all } \mathbf{x} \in \Omega \tag{5.20}$$

What the theorem is saying is that the entire space of continous functions can be spanned with a linear combination of the function $\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$. Note that we are always using the same $\mathbf{W}$ and $\mathbf{b}$, so we are only taking linear combinations of one $\sigma(\cdot)$, there is no composition and thus the network in the theorem has just one hidden layer. For large enough q, the training error can be made *arbitrarily small*.
UATs exist for other activations like ReLUs and locally bounded non-polynomials. The problem with these theorems is that the proofs are *not constructive*, and thus do not say how to compute the weights to reach a desired accuracy.
Some theorems also give bounds for the *width* q ("number of neurons"), while some show universality for > 1 *layers* (deep networks).
Nevertheless, in general, we deal with nonconvex functions. Empirical results show that large q combined gradient descent leads to very good approximations.

5.3 Training

Given a MLP with training set $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{n}$

$$g_{\mathbf{\Theta}}(\mathbf{x}_i) = (\sigma \circ f_{\mathbf{\Theta}_n}) \circ (\sigma \circ f_{\mathbf{\Theta}_{n-1}}) \circ \cdots \circ (\sigma \circ f_{\mathbf{\Theta}_1})(\mathbf{x}_i) = \mathbf{y}_i \tag{5.21}$$

and a suitable loss function, such as the Mean Squared Error loss defined as

$$\ell_{\mathbf{\Theta}}(\{\mathbf{x}_i, \mathbf{y}_i\}) = \frac{1}{n} \sum_{i=1}^{n} \|\mathbf{y}_i - g_{\mathbf{\Theta}}(\mathbf{x}_i)\|_2^2, \tag{5.22}$$

we have seen that solving for the weights $\mathbf{\Theta}$ is referred to as *training*.
In general, the loss will be a *non-convex* function of the weights $\mathbf{\Theta}$. As we have seen, the following *special cases* are convex:

- One layer, no activation, MSE loss ($\Rightarrow$ linear regression).
- One layer, sigmoid activation, logistic loss ($\Rightarrow$ logistic regression).

We have also seen that training is usually performed using gradient descent-like algorithms, that

require the computation of gradients $\nabla \ell_{\boldsymbol{\Theta}}$. For the basic MSE, this means

$$\nabla \ell_{\boldsymbol{\Theta}}(\{\mathbf{x}_i, \mathbf{y}_i\}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\Theta}} \|\mathbf{y}_i - g_{\boldsymbol{\Theta}}(\mathbf{x}_i)\|_2^2 \tag{5.23}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\Theta}} \|(\mathbf{y}_i - (\sigma(f_{\boldsymbol{\Theta}_n}(\sigma(f_{\boldsymbol{\Theta}_{n-1}}(\cdots(\sigma(f_{\boldsymbol{\Theta}_1}(\mathbf{x}_i))\cdots)))))))\|_2^2 \tag{5.24}$$

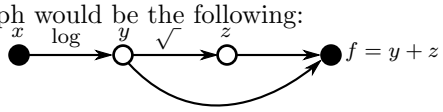
Of course for different loss functions and different models we will have different expressions for the gradient; however, computing the gradients *by hand* would be infeasible.
The derivatives could be approximated numerically by approximating the limit of the ratio, but doing so requires $O(\#\text{weights})$ evaluations of $\ell_{\boldsymbol{\Theta}}$. The gradient can be computed automatically using the *chain rule*, but this is still sub-optimal. What we want to do is automatize this *computational step* efficiently, and this is done by *automatic differentiation*.

5.3.1 Computational graphs

Consider a generic function $f : \mathbb{R} \to \mathbb{R}$: its *computational graph* is a directed acyclic graph representing the computation of $f(x)$ with *intermediate variables*.
Example:

$$f(x) = \log x + \sqrt{\log x} \tag{5.25}$$

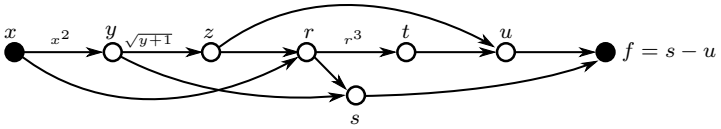
The resulting computational graph would be the following:



Example:

$$f(x) = \frac{\log\left(x + \sqrt{x^2+1}\right)}{x^2} - \frac{\log^3(x + \sqrt{x^2+1})}{\sqrt{x^2+1}} \tag{5.26}$$

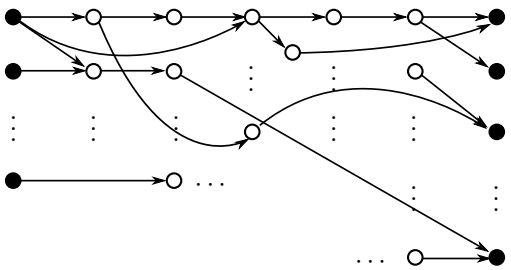
The resulting computational graph would be the following:



The graph is constructed programmaticaly, using elementary functions as building blocks. For example in the previous example the edge $x \to z$ would be defined as:

$$z = sqrt(sum(square(x), 1)). \tag{5.27}$$

For *high-dimensional* input/output, the graph may be more complex, since the output vector has several components to be defined, and the components of the input vector can take part in their computation in various intricated ways:



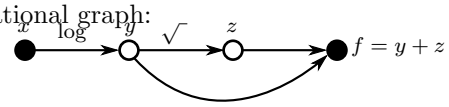
Keep in mind that the computational graphs can become arbitrarily complex. However, with this formulation the evaluation of $f(x)$ corresponds always to a *forward traversal* of the graph, in which the input variables hold actual values that are used to compute the values for the intermediate variables and eventually the output variables.

5.3.2 Automatic differentiation: forward mode

The first way we can compute derivatives is by traversing the computational graph in *forward mode*, that is, from input to output. Assume we have the following function

$$f(x) = \log x + \sqrt{\log x} \tag{5.28}$$

with the corresponding computational graph:



So what we do is basically apply the *chain rule* in a recursive way, starting from the input and going forward. Note that at each node the derivative is computed with respect to the input (x in this case), not with respect to the preceding node.

$$\begin{aligned}
\frac{\partial x}{\partial x} &= 1 \\
\frac{\partial y}{\partial x} &= \frac{\partial y}{\partial x}\frac{\partial x}{\partial x} = \frac{\partial \log x}{\partial x}\frac{\partial x}{\partial x} = \frac{1}{x}\frac{\partial x}{\partial x} \\
&= \frac{1}{x} \\
\frac{\partial z}{\partial x} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial \sqrt{y}}{\partial y}\frac{\partial y}{\partial x} = \frac{1}{2\sqrt{y}}\frac{\partial y}{\partial x} \\
&= \frac{1}{2\sqrt{y}}\frac{1}{x} \\
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial (y+z)}{\partial y}\frac{\partial y}{\partial x} + \frac{\partial (y+z)}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} + \frac{\partial z}{\partial x} \\
&= \frac{1}{x} + \frac{1}{2\sqrt{y}}\frac{1}{x}
\end{aligned} \tag{5.29}$$

This way, going forward we can compute both $f(x)$ and its derivative, as it is computed step by step from the input to the output. So, if we assume that each partial derivative is a primitive accessible in closed form and can be computed in $O(1)$ we have that

$$\text{cost of computing } \tfrac{\partial f}{\partial x}(x) \;=\; \text{cost of computing } f(x).$$

However, if the input is high-dimensional, i.e. $f : \mathbb{R}^p \to \mathbb{R}$:

$$\text{cost of computing } \nabla f(\mathbf{x}) \;=\; p \;\times \text{ cost of computing } f(\mathbf{x})$$

since partial derivatives must be computed w.r.t. each input dimension. Now, since p can be in the order of millions, this is definetely infeasible.

Remark:

$$\underset{\text{(e.g. autograd)}}{\text{Automatic differentiation}} \neq \underset{\text{(e.g. Mathematica)}}{\text{Symbolic differentiation}}$$

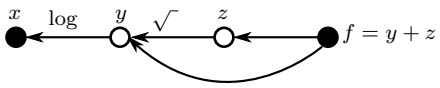
In the latter, a mathematical expression is given in input and a mathematical expression is returned, while in the former the *values* are computed to generate numerical evaluations of the derivatives.

5.3.3 Automatic differentiation: reverse mode

In *reverse mode*, we compute all the partial derivatives $\frac{\partial f}{\partial z}, \dots, \frac{\partial f}{\partial x}$ with respect to the *inner nodes*. To stress the difference, remind that in the forward mode we computed the derivatives of each node with respect to the input, so the derivative of the last node was the derivative of f wrt to x.
Let's consider again the same function

$$f(x) = \log x + \sqrt{\log x}$$

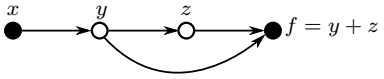
with the same computation graph:



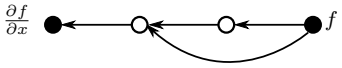
Then automatic differention in reverse mode would be:

$$\begin{aligned}
\frac{\partial f}{\partial f} &= 1 \\
\frac{\partial f}{\partial z} &= \frac{\partial f}{\partial f}\frac{\partial f}{\partial z} = \frac{\partial f}{\partial f}\frac{\partial (y+z)}{\partial z} = \frac{\partial f}{\partial f} \\
&= 1 \\
\frac{\partial f}{\partial y} &= \frac{\partial f}{\partial z}\frac{\partial z}{\partial y} + \frac{\partial f}{\partial f}\frac{\partial f}{\partial y} = \frac{\partial f}{\partial z}\frac{\partial \sqrt{y}}{\partial y} + \frac{\partial f}{\partial f}\frac{\partial (y+z)}{\partial y} = \frac{\partial f}{\partial z}\frac{1}{2\sqrt{y}} + \frac{\partial f}{\partial f} \\
&= \frac{1}{2\sqrt{y}} + 1 \\
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial f}{\partial y}\frac{\partial \log x}{\partial x} = \frac{\partial f}{\partial y}\frac{1}{x} \\
&= \frac{1}{x}\left(\frac{1}{2\sqrt{y}} + 1\right).
\end{aligned} \tag{5.30}$$

Notice that, as expected, the resulting expression for $\frac{\partial f}{\partial x}$ in both eq. (5.29) eq. (5.30) is the same. There are however differences between the two procedures.
First, notice that in eq. (5.30) some expressions are defined in terms of *internal nodes*, so reverse mode requires computing the values of the internal nodes first. This is not necessary in forward mode since it takes place with the same order the graph traversal, therefore their values is already computed.
So first a *forward pass* must be done to evaluate all the interior nodes $y, z, \dots$



then a *backward pass* must be done to compute the *derivatives*. Note that the forward pass is *not* forward-mode autodiff, since we are only computing *function values*.



5.3.4 Automatic differentiation: complexity

Suppose we have a function

$$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m, \tag{5.31}$$

defined as a composition of functions

$$\mathbf{f} = \mathbf{f}_l \circ \mathbf{f}_{l-1} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1 \tag{5.32}$$

such that

$$\begin{aligned} \mathbf{f}_1 &: \mathbb{R}^n \to \mathbb{R}^{d_1}, \\ \mathbf{f}_k &: \mathbb{R}^{d_{k-1}} \to \mathbb{R}^{d_k}, \\ \mathbf{f}_l &: \mathbb{R}^{d_{l-1}} \to \mathbb{R}^m. \end{aligned} \tag{5.33}$$

This function will have a very straightforward computation graph, a chain going from $\mathbf{f}_1$ up to $\mathbf{f}_l$, computing "hidden" variables $\mathbf{h}_1, \dots, \mathbf{h}_{l-1}$ along the way. Note that since it is $\mathbf{h}_i = \mathbf{f}_i(\mathbf{h}_{i-1})$ the following expressions are equivalent

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{h}_{i-1}} \equiv \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}. \tag{5.34}$$

Let $\mathbf{x} \in \mathbb{R}^n$ be the input to the computation graph, and suppose we want to compute

$$\underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \mathbf{J}_\mathbf{x}\mathbf{f}}_{\text{Jacobian}} = \underbrace{\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}}_{(m \times n)}. \tag{5.35}$$

where note that f_i is the i-th component of $\mathbf{f}$ while $\mathbf{f}_i$ is the i-th function in the composition. By the chain rule this would be

$$\begin{aligned} \mathbf{J}_\mathbf{x}\mathbf{f} &= \mathbf{J}_{\mathbf{h}_{l-1}}[\mathbf{f}_l]\ \mathbf{J}_\mathbf{x}[\mathbf{f}_{l-1} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1] \\ &= \mathbf{J}_{\mathbf{h}_{l-1}}[\mathbf{f}_l]\ \mathbf{J}_{\mathbf{h}_{l-2}}[\mathbf{f}_{l-1}]\ \mathbf{J}_\mathbf{x}[\mathbf{f}_{l-2} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1] \\ &\ \ \vdots \\ &= \mathbf{J}_{\mathbf{h}_{l-1}}[\mathbf{f}_l]\ \mathbf{J}_{\mathbf{h}_{l-2}}[\mathbf{f}_{l-1}]\ \dots\ \mathbf{J}_{\mathbf{h}_1}[\mathbf{f}_2]\ \mathbf{J}_\mathbf{x}[\mathbf{f}_1] \end{aligned} \tag{5.36}$$

in which each computation of $\mathbf{J}_\mathbf{x}[\mathbf{f}] = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is defined as in eq. (5.35) and thus returns a matrix, called again Jacobian

$$\mathbf{J}_\mathbf{x}\mathbf{f} = \mathbf{J}_{l-1}\ \mathbf{J}_{l-2}\ \dots\ \mathbf{J}_2\ \mathbf{J}_1 \tag{5.37}$$

where now the subscript is not the variable wrt which we are differentiating any longer but reminds us of the correct matrix.
This is from an analytical point of view, and both procedures must perform this computation to arrive at the correct result; however, the different order in which they perform the several matrix multiplications leads to different computational cost.

- Forward-autodiff will perform computations going forward, so it will first compute

$$\mathbf{J}_1 = \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}},$$

then

$$\mathbf{J}_2\mathbf{J}_1 = \frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}$$

then

$$\mathbf{J}_3(\mathbf{J}_2\mathbf{J}_1) = \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}$$

so that finally we will have

$$\mathbf{J}_{l-1}(\mathbf{J}_{l-2}(\cdots(\mathbf{J}_3(\mathbf{J}_2\mathbf{J}_1)))). \tag{5.38}$$

- Reverse-autodiff will instead perform computations going backward, so it will first compute

$$\mathbf{J}_{l-1} = \frac{\partial \mathbf{f}_l}{\partial \mathbf{h}_{l-1}},$$

then

$$\mathbf{J}_{l-1}\mathbf{J}_{l-2} = \frac{\partial \mathbf{f}_l}{\partial \mathbf{h}_{l-1}}\frac{\partial \mathbf{f}_{l-1}}{\partial \mathbf{h}_{l-2}},$$

then

$$(\mathbf{J}_{l-1}\mathbf{J}_{l-2})\mathbf{J}_{l-3} = \frac{\partial \mathbf{f}_l}{\partial \mathbf{h}_{l-1}}\frac{\partial \mathbf{f}_{l-1}}{\partial \mathbf{h}_{l-2}}\frac{\partial \mathbf{f}_{l-2}}{\partial \mathbf{h}_{l-3}}$$

so that finally we will have

$$((((\mathbf{J}_{l-1}\mathbf{J}_{l-2})\mathbf{J}_{l-3})\cdots)\mathbf{J}_2)\mathbf{J}_1. \tag{5.39}$$

Let's compare the computational cost of such procedures. We will consider as "elementary operation" the multiplication involved in each *dot product* required to compute one entry of a matrix, i.e. to compute

$$\begin{bmatrix} \mathbf{a}^\top\mathbf{b} & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} — & \mathbf{a}^\top & — \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix} \begin{bmatrix} | & \vdots & \vdots \\ \mathbf{b} & \vdots & \vdots \\ | & \vdots & \vdots \end{bmatrix}, \tag{5.40}$$

if $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$, it would require d operations.

- **Forward-autodiff**.
  $\mathbf{J}_1$ is a $(d_1 \times n)$ matrix, while $\mathbf{J}_2$ is a $(d_2 \times d_1)$ matrix, so $\mathbf{J}_2\mathbf{J}_1$ will be a $(d_2 \times n)$ matrix.
  $\mathbf{J}_3$ is a $(d_3 \times d_2)$ matrix, so $\mathbf{J}_3(\mathbf{J}_2\mathbf{J}_1)$ will be a $(d_3 \times n)$ matrix.

  Notice how the matrix being computed always has n columns, since we expand leftward and the rightmost matrix in the computation is always $\mathbf{J}_1$. In fact the generic computation will be

$$\mathbf{J}_{1:k} = \mathbf{J}_k\ \mathbf{J}_{1:k-1} = \mathbf{J}_k\ (J_{k-1}(J_{k-2}(\cdots(\mathbf{J}_3(\mathbf{J}_2\mathbf{J}_1)))) \tag{5.41}$$

  in which $\mathbf{J}_k$ will be a $(d_k \times d_{k-1})$ matrix and $\mathbf{J}_{1:k-1}$ will be a $(d_{k-1} \times n)$ matrix, so the result will be a $(d_k \times n)$ matrix.

  Each one of these computations will thus require to fill a $(d_k \times n)$ matrix with dot products between vectors of dimension d_{k-1}, thus resulting in a total complexity of

$$\sum_{k=2}^{l-1} d_k\ n\ d_{k-1} + \underbrace{m\ n\ d_{l-1}}_{\text{last step}} \approx n \sum_{k=1}^{l-2} d_k d_{k+1} \tag{5.42}$$

  by neglecting the last computation on the "output" function.

- **Backward-autodiff**.
  We proceed with an identical reasoning as before. Now we start from $\mathbf{J}_{l-1}$ that is a $(m \times d_{l-1})$ matrix, and we expand rightward so the leftmost matrix in the generic computation will always be $\mathbf{J}_{l-1}$. In fact the generic computation will be

  $$\mathbf{J}_{l-1:k} = \mathbf{J}_{l-1:k+1}\ \mathbf{J}_k = ((\mathbf{J}_{l-1}\mathbf{J}_{l-2})\mathbf{J}_{l-3})\cdots)\mathbf{J}_{k+2})\mathbf{J}_{k+1})\ \mathbf{J}_k \tag{5.43}$$

  in which $\mathbf{J}_{l-1:k+1}$ will be a $(m \times d_{k+1})$ matrix and $\mathbf{J}_k$ will be a $(d_{k+1} \times d_k)$ matrix, so the result will be a $(m \times d_k)$ matrix.

  Again, each one of these computations will therefore require to fill a $(m \times d_k)$ matrix with dot products between vectors of dimension d_{k+1}, thus resulting in a total complexity of

  $$\sum_{k=l-1}^{2} m\ d_k\ d_{k+1} + \underbrace{m\ n\ d_1}_{\text{last step}} \approx m \sum_{k=2}^{l-1} d_k d_{k+1} \tag{5.44}$$

  by neglecting the last computation on the "input" function.

Looking at the results of this analysis, we can make some observations.

- Forward-autodiff requires n traversals (notice how the sum index spans the layers of the network) to compute the Jacobian. Forward-autodiff can compute the derivatives of *all the output variables wrt to a single input variable* in a single pass. Since there are n input variables, the procedure requires n traversals.

  Intuitively, recall that it computes derivatives going forward in the graph, and a single input variable influences (in the computation graph has paths leading to) in principle every output variable, so all of them will be reached in a single pass. At each step in the traversal it looks at the variables reached so far and uses them in all the computations in which they are required to go forward, accumulating the derivatives always wrt to the input variable under consideration.

- On the other hand, reverse-autodiff requires m traversals (again notice the sum index, that in the first equation has been written going "backward" to highlight the backward spanning of the network layers) to compute the Jacobian. Reverse-autodiff can compute the derivatives of *all the input variables wrt to a single output variable* in a single pass. Since there are m output variables, the procedure requires m traversals.

  Intuitively, recall that it computes derivatives going backward in the graph, and a single output variable is influenced (in the computation graph has paths coming from) in principle every input variable, so all of them will be reached in a single pass. At each step in the traversal it looks at the variables reached so far and looks at the variables needed for computing them to go backward, accumulating derivatives always of the output variable under consideration.

So, the complexity of the two approaches is indeed influenced by the dimensionality of the domain and the codomain of the function f under consideration. This means that:

- forward-autodiff is best suited when derivatives are needed for functions $\mathbf{f} : \mathbb{R} \to \mathbb{R}^m$;
- reverse-autodiff is best suited in the other extreme case of needing derivatives for functions $f : \mathbb{R}^n \to \mathbb{R}$.

5.3.5 Backpropagation

We call *back-propagation* (or *backprop* in short) the reverse mode automatic differentiation applied to deep neural networks.
Why do we choose reverse-autodiff? We will use what we have seen so far but in the deep learning context.
When training a deep neural network, we are minimizing a loss function with respect to the $p \gg 1$ *weights* (or *parameters*) of the network, represented by a set of matrices

$$\mathbf{W} = \{W^{(i)} \in \mathbb{R}^{p_i}, i \in \{1, \dots, t-1\}\}. \tag{5.45}$$

In particular, the newtwork will be a function

$$\begin{aligned} \mathbf{f}(\mathbf{x}; \mathbf{W}) &= \mathbf{f}_{t-1}(\mathbf{f}_{t-2}(\cdots(\mathbf{f}_2(\mathbf{f}_1(\mathbf{x}; \mathbf{W}^{(1)}); \mathbf{W}^{(2)})); \mathbf{W}^{(t-2)}); \mathbf{W}^{(t-1)}) \\ \mathbf{f} &= \mathbf{f}_{t-1} \circ \mathbf{f}_{t-2} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1 \end{aligned} \tag{5.46}$$

with each function $\mathbf{f}_i$ in the composition parametrized by $\mathbf{W}^{(i)}$.
Also, the loss will be some error criterion ϵ, and inside the loss there will be the network:

$$\ell = \epsilon(\mathbf{f}_{t-1} \circ \mathbf{f}_{t-2} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1). \tag{5.47}$$

We have seen that minimizing the loss function requires the computation of its *gradient* $\boldsymbol{\nabla}_{\mathbf{W}} \ell$, and that this computation can be decomposed by the chain rule in the computation of several intermediate derivatives. We have

$$\begin{aligned} \boldsymbol{\nabla}_{\mathbf{W}} \ell &= \boldsymbol{\nabla} \epsilon \; \mathbf{J}_{\mathbf{W}}(\mathbf{f}_{t-1} \circ \mathbf{f}_{t-2} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1) \\ &= \boldsymbol{\nabla} \epsilon \; \mathbf{J}_{t-1} \mathbf{J}_{t-2} \dots \mathbf{J}_2 \mathbf{J}_1 \end{aligned} \tag{5.48}$$

in which we denote by $\mathbf{J}_k$ the Jacobian at layer k. In the following we will neglect $\boldsymbol{\nabla} \epsilon$ since as we have seen the complexity lies in the function composition.
Now, recall our previous analysis on the complexity of the two autodiff procedures, and that the loss function is a function:

$$\ell : \mathbb{R}^p \to \mathbb{R}^1 \tag{5.49}$$

with $p = p_1 + \cdots + p_{t-1}$.

- Forward-mode autodiff scales linearly with the dimension of the domain of the function, p in this case.
- Reverse-mode autodiff instead scales linearly with the dimension of the codomain of the function, 1 (!) in this case.

So, as p can be in the order of millions, the latter approach is the way to go.

5.3.6 Observations

- Evaluating $\nabla \ell$ with backprop is as fast as evaluating ℓ. In fact, the forward pass is exactly the evaluation of ℓ via a forward traversal of the graph, and we have seen that the complexity of the backward pass for vector-to-scalar functions coincides with a simple traversal.
- Back-propagation is not just the chain rule, as some mistakenly believe. In fact, back-propagation *uses* the chain rule within some more *sophisticated pipeline*, comprised of a forward pass with *intermediate variables* stored to then compute a backward pass. It is more precise to say that backprop is a *computational* technique.

- One does not backprop "through the network", but rather through the computational graph of the loss.
- The loss of a MLP will be *non-convex* in general, presenting multiple *local minima*; which of these is reached depends on the *weight initialization*. In practice, reaching the global optimum usually leads to overfitting, since it would mean that we are fitting the function too closely to the data, accounting for the noise.
- The loss of a MLP will be *non-differentiable* in general; for example, the ReLU is not differentiable at zero. What happens is that software implementations usually return one of the one-sided derivatives. Nevertheless, *numerical issues* are always behind the corner.
- Lastly, keep in mind that effectively training a deep network is far from a solved problem.

Chapter 6

Convolutional neural networks

In the previous chapter we have presented the Multi-Layer Perceptron, what most people refer to when speaking about a Neural Network in general. These are *deep* networks, since they are made up of several layers, and are *feed-forward* networks, since the data progresses through the network from the input layer to the output layer in a straight-forward way, undergoing transformations at each layer in the process.

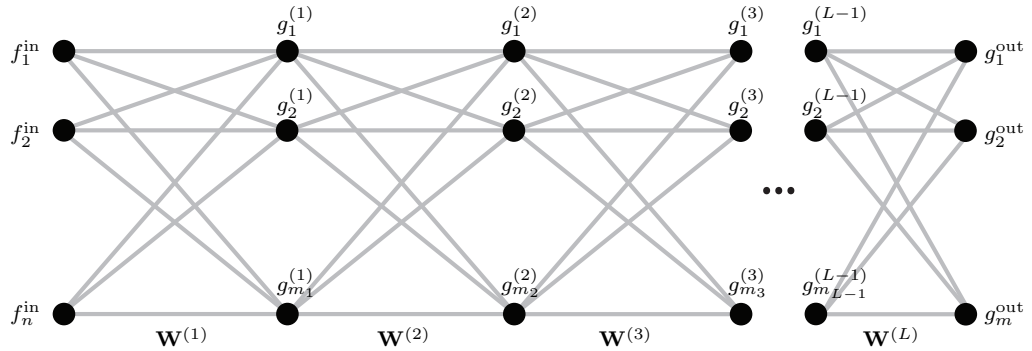


Figure 6.1: Deep feed-forward neural network consisting of L layers.

Let $\mathbf{g}^{(k)} = \begin{pmatrix} g_1^{(k)} & \dots & g_{m_k}^{(k)} \end{pmatrix}^\top$ be the vector output of the k-th layer of the network. The layer k performs a transformation on the output of the previous layer to compute

$$\mathbf{g}^{(k)} = \sigma(\mathbf{W}^{(k)}\mathbf{g}^{(k-1)}) \tag{6.1}$$

where the ℓ component is

$$g_\ell^{(k)} = \sigma\left(\sum_{\ell'=1}^{m_{k-1}} g_{\ell'}^{(k-1)} w_{\ell,\ell'}^{(k)}\right) \quad \begin{matrix} \ell = 1, \dots, m_k \\ \ell' = 1, \dots, m_{k-1} \end{matrix} \tag{6.2}$$

where $\sigma(x)$ is the *activation* function, *e.g.* the *Rectified Linear Unit* (ReLU):

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

All these layers have trainable parameters, that get adjusted via an optimization algorithm like *Stochastic Gradient Descent* to minimize a *loss function*. These parameters are the weights

$$\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}$$

including the biases.
With this architecture, the network output is

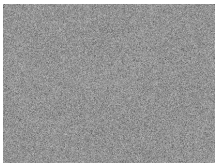
$$\mathbf{g}^{\text{out}} = \sigma\left(\mathbf{W}^{(L)}\left(\dots\left(\mathbf{W}^{(2)}\sigma\left(\mathbf{W}^{(1)}\mathbf{f}^{\text{in}}\right)\right)\dots\right)\right). \tag{6.3}$$

In principle, this architecture is as general as it gets: deep feed-forward neural networks are *provably universal*, meaning that provided enough units, they can approximate any function with any desired accuracy. However, this comes with a price:

- We can make them *arbitrarily complex*;
- The number of *parameters* increases very rapidly and can get huge;
- The two points above combined make it so these network can become very difficult to *optimize*;
- Even then, with this architecture is very difficult to achieve *generalization*, since often they become *too powerful* and manage to represent perfectly the data, overfitting and losing generalization power.

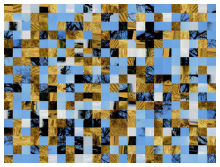
6.1 Need for Priors

A partial remedy for the problems above comes from the data itself: the *priors*. A neural network is a blank sheet before it gets fed data to fit, it has no idea how this data is structured or should be structured. Therefore, we look for "universal" priors, ideally task-independent to some extent. A key insight is that data often carries *structural priors* in terms of repeating patterns, compositionality, locality, self-similarity, that we want to exploit to make it easier for neural networks to accurately represent the domain of interest and perform well at the task at hand.



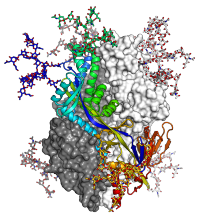
(a) No structure whatsoever, no priors, just noise.

(b) We have already discussed some priors of this image.



(c) Real world images are rarely puzzles in disarray like this...

(d) ...but rather exhibit self-similarity and locality properties like this.



(e) Proteins are all about structure.

Figure 6.2: Examples of *structure*; we want to take advantage of this structure by means of *priors*.

Let's see some of these priors.

6.1.1 Self-similarity

Data tends to be *self-similar* across the domain, meaning that one can identify building blocks that are repeated almost identically, to build up the data.

Figure 6.3: Self-similarity means less information needed to be represented by the network. Self-similarity also means data is *predictable*: we expect the fence and the vegetation to continue behind the eagle, lazily repeating itself, and so does the network, that is able to remove it from the image and fill in the gap in a credible way.

6.1.2 Translation invariance

Translation does not change the information content of an image, therefore it is desirable to enforce *translation invariance*. What this means is that if two pieces of data are identical up to a translation (*e.g.*, an object is shifted in an image), then we want our networks to produce identical outputs.

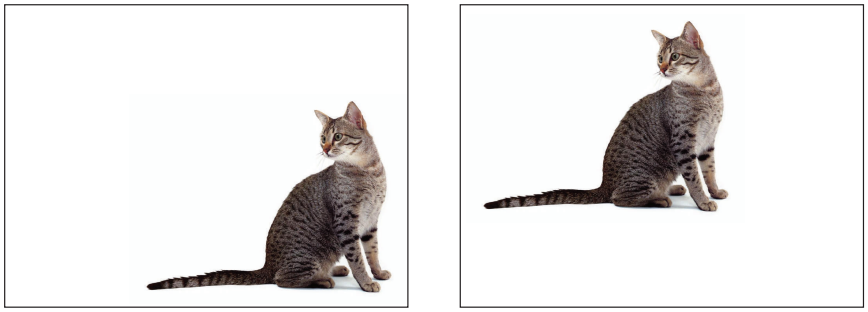


Figure 6.4: A cat is still a cat regardless of where it is located in an image.

Formally, we define the *translation operator* in the following way:

$$\mathcal{T}_v f(x) = f(x - v) \tag{6.4}$$

where f is the function encoding the data. For instance, if we had an image, x would be the two-dimensional vector of pixel coordinates, and $f(x)$ would be a three-dimensional vector of RGB values. So, we are defining an operator that transforms data $f(x)$ such that data at position $x - v$ ends up ad position x. With this definition, translation invariance is defined as:

$$y(\mathcal{T}_v f) = y(f) \quad \forall f, \mathcal{T}_v \tag{6.5}$$

where y is a classification functional such as our network.

6.1.3 Other invariances

Many other types of invariance may be desirable; for example, *deformation* invariance.

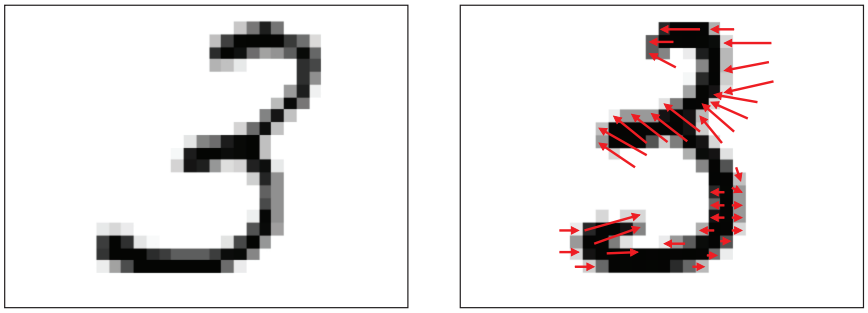


Figure 6.5: A digit is still a digit regardless of how much (within some degree) deformed it appears in images of handwritten text.

Consider the *warping operator* $\mathcal{L}$:

$$\mathcal{L}_\tau f(x) = f(x - \tau(x)), \tag{6.6}$$

where τ is a *deformation field*. What it does is similar to the translation operator, meaning that the value of the transformed function $f(\cdot)$ at a point x is computed by means of a shift; however,

the amount of the shift is not a fixed offset v, but depends on the local behavior of deformation field τ at that point. With this definition, the desirable invariance would be:

$$|y(\mathcal{L}_\tau f) - y(f)| \approx \|\nabla \tau\| \quad \forall f, \tau \tag{6.7}$$

where again y is a classification functional such as our network, and we are enforcing that the prediction on warped data should be different from the prediction on non-warped data by an amount proportional to the amount of deformation present.
We might also desire invariance to *partiality* and *isometric deformations*.

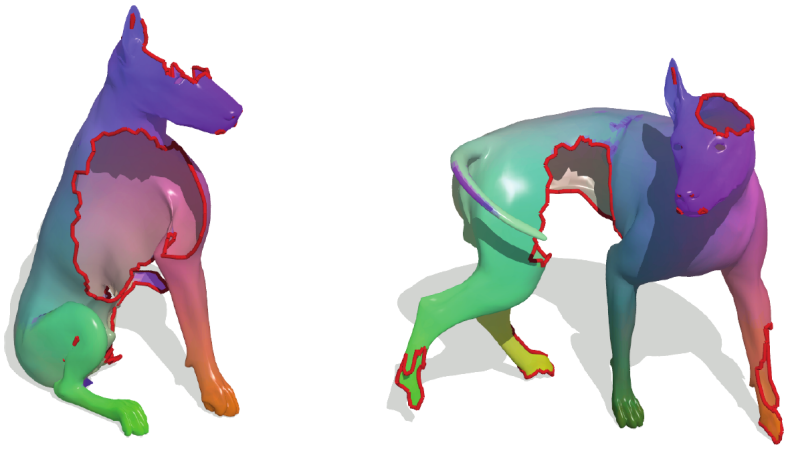


Figure 6.6: Even if these two models are displaced (deformed) differently, and miss some parts (therefore are partials), we still recognize both of them as dogs.

6.1.4 Hierarchy and compositionality

Translational invariance is one of the most common and therefore most desirable invariances. In particular, is desirable *across multiple scales*, leading to *compositionality*. In a hierarchical way, we expect the data (we will concentrate on images in this chapter) to be able to be decomposed in *local features* at each scale, invariant of their location in the image:

$$z(\mathcal{T}_v p) = z(p) \quad \forall p, \mathcal{T}_v \tag{6.8}$$

where p are image patches of variable size.

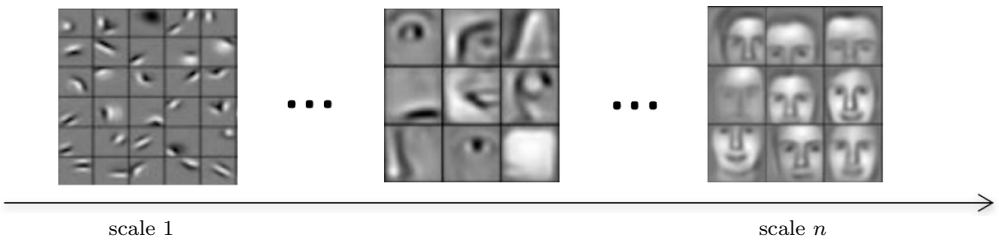


Figure 6.7: A human face is still a human face even when translated, and so is a nose or an eye, and so are the edges that make up each of those, provided that when arranged together, the local features at a lower scale *compose* a correct local feature at a higher scale.

6.2 Convolution

We have seen that data is often composed of *hierarchical*, *local*, *shift-invariant* patterns, and we want to exploit that as a prior. A particular class of neural networks, called *Convolutional Neural Networks* (CNNs) exploit this fact directly, through the distinctive operation that it applies to data: *convolution*.
Convolution is classically defined as an operation on two functions of a real-valued argument, that gives back another function. The term convolution refers to both the result function and to the process of computing it. Given two functions $f, g : \mathbb{R} \to \mathbb{R}$ their *convolution* is a function:

$$\underbrace{(f \star g)(t)}_{feature\ map} = \int_{-\infty}^{+\infty} f(\tau)\, \underbrace{g(t-\tau)}_{kernel}\, d\tau. \tag{6.9}$$

Intuitively, the convolution formula can be described as a weighted average of the function $f(\tau)$ at the point t where the weighting is given by $g(-\tau)$ simply shifted by amount t. As t changes, the weighting function emphasizes different parts of the input function.
Operatively, computing convolution means performing the following steps:

1. Express each function in terms of a dummy variable τ;
2. Reflect one of the functions: $g(\tau) \to g(-\tau)$;
3. Add a time-offset, t, which allows $g(t-\tau)$ to slide along the τ-axis;
4. Start t at $-\infty$ and slide it all the way to $+\infty$. Wherever the two functions intersect, find the integral of their product, which means consider the area under the curve of their intersection.

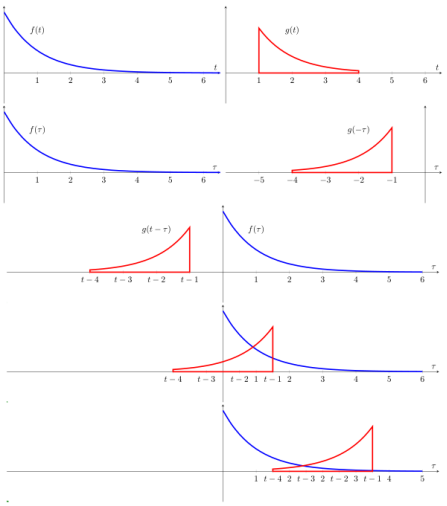


Figure 6.8: Operative illustration of convolution.

In the following example, we have a red-colored "pulse", $g(\tau)$, and a blue-colored pulse $f(\tau)$. The amount of yellow is the area of the product $f(\tau) \cdot g(t-\tau)$, computed by the convolution integral. The animation is created by continuously changing t and recomputing the integral. The result

(shown in black) is a function of t, not of τ, but is plotted on the same axis as τ, for convenience and comparison.

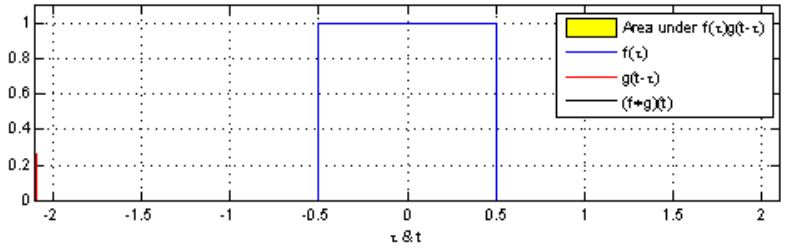


Figure 6.9: Convolution of two identical "pulse" functions. If the animation is not displayed correctly, try to enable JavaScript in your PDF reader or use Adobe Reader which has the correct extensions already enabled.

6.2.1 Properties

The convolution operator has a number of nice properties.

Commutativity

Convolution is *commutative*

$$(f \star g)(x) = \int_{-\infty}^{+\infty} f(t)g(x-t)dt \overset{z:=x-t}{=\!=\!\downarrow} \int_{-\infty}^{+\infty} f(x-z)g(z)dz = (g \star f)(x). \tag{6.10}$$

Shift-equivariance

Convolution is *shift-equivariant*

$$f(x - x_0) \star g(x) = (f \star g)(x - x_0). \tag{6.11}$$

What this means is that if you apply convolution with a certain kernel to two identical entities, one shifted and the other one not shifted, you get the same feature maps, but one is shifted and the other one is not. In fact, equivariance is a *defining property* of convolutions, meaning an operator that has this property can be shown to behave like a convolution. This is good news, since it seems like convolutions are built to support one of the priors we identified.

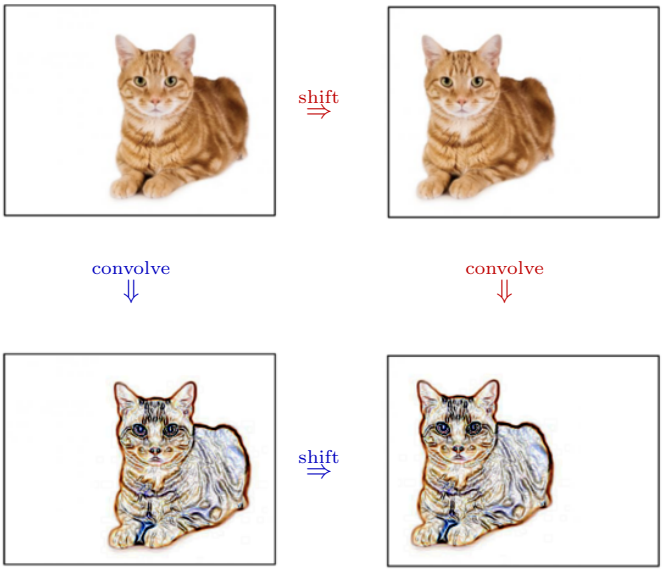


Figure 6.10: Convolution can be applied to images, as we will see shortly, and exhibits shift-equivariance.

Linearity

Convolution is a *linear* operator. If we denote with $\mathcal{G}$ the convolution operator, we have

$$\mathcal{G}f(x) = (f \star g)(x) = \int_{-\infty}^{+\infty} f(t) \underbrace{g(x-t)}_{kernel} dt.$$

It is easy to show that $\mathcal{G}$ is linear, since it is a direct consequence of the linearity of the integral operator upon which it is built; in fact, homogeneity holds

$$\mathcal{G}(\alpha f(x)) = \alpha \int_{-\infty}^{+\infty} f(t)g(x-t)dt = \alpha\, \mathcal{G}f(x) \tag{6.12}$$

and additivity as well

$$\begin{aligned}
\mathcal{G}(f+h)(x) &= \int_{-\infty}^{+\infty} (f+h)(t)g(x-t)dt && (6.13)\\
&= \int_{-\infty}^{+\infty} (f(t)+h(t))g(x-t)dt && (6.14)\\
&= \int_{-\infty}^{+\infty} f(t)g(x-t)dt + \int_{-\infty}^{+\infty} h(t)g(x-t)dt && (6.15)\\
&= \mathcal{G}f(x) + \mathcal{G}h(x). && (6.16)
\end{aligned}$$

From this, if we consider the translation (shift) operator $\mathcal{T}$, translation equivariance can then be formulated as

$$\mathcal{G}(\mathcal{T}f) = \mathcal{T}(\mathcal{G}f) \tag{6.17}$$

i.e the convolution and translation operators commute.

6.2.2 Discrete Convolution

We have seen that convolution deals with continuous, real-valued functions. But our data (like images) live in the discrete, vector-valued domain. The ideas behind convolution can be adapted to the *discrete* setting, in which we define the *convolution sum*:

$$(\mathbf{f} \star \mathbf{g})[n] = \sum_{k=-\infty}^{\infty} \mathbf{f}[k]\mathbf{g}[n-k] \tag{6.18}$$

where f, g are no longer functions but rather *sequences*.

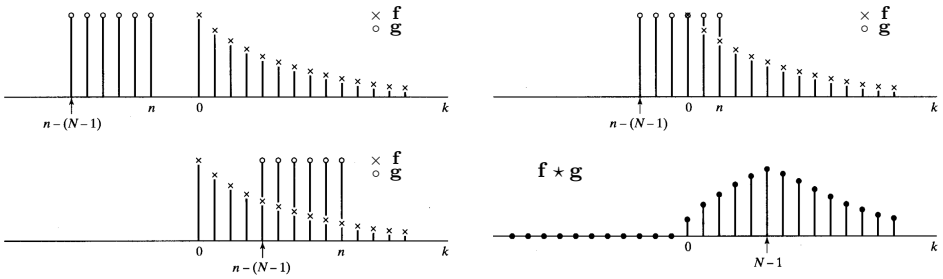


Figure 6.11: Visualization of discrete convolution on two sequences f, g.

In the example above, f was *zero-padded* in order for the products to be well defined for all shifts (*boundary conditions*). If the sequences f, g are finite (as it is usually the case with real data), then the convolution operator can be encoded as a *Toepliz matrix* (or *circulant* matrix) as follows:

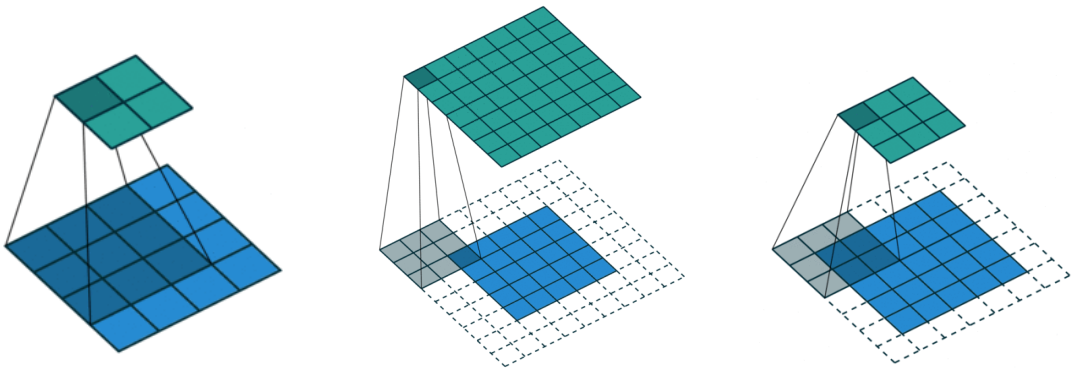
$$\mathbf{f} \star \mathbf{g} = \begin{pmatrix} g_1 & g_2 & \cdots & \cdots & g_n \\ g_n & g_1 & g_2 & \cdots & g_{n-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ g_3 & g_4 & \cdots & g_1 & g_2 \\ g_2 & g_3 & \cdots & \cdots & g_1 \end{pmatrix} \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix}. \tag{6.19}$$

This can be extended to 2-dimensional data. For instance we can consider RGB images as sequences $f : \mathbb{N}^2 \to \mathbb{R}^3$. Then we can consider each channel independently, thus considering three sequences $f : \mathbb{N}^2 \to \mathbb{R}$ over which we define the following $2D$ convolution sum:

$$(\mathbf{f} \star \mathbf{g})[m, n] = \sum_{k} \sum_{\ell} \mathbf{f}[k, \ell]\, \mathbf{g}[m-k, n-\ell]. \tag{6.20}$$

Figure 6.12: Visual interpretation of $2D$ convolution of an image with a kernel as the action of *sliding* a *moving window* onto the image.

We can further generalize convolution: for functions $f : \mathbb{R}^k \to \mathbb{R}^m$ defined on *Euclidean domains*, convolution is well-defined up to appropriate *boundary conditions*. In practice, convolution is often replaced by other operations with similar properties (locality, compositionality, etc.), as we'll see.



(a) No padding: the convolution kernel is directly applied within the boundaries of the underlying function (an image in this example). The result of the convolution is a smaller image.

(b) Full zero-padding: the domain is enlarged and padded with zeroes. The convolution kernel is applied within the (now larger) boundaries. The result of the convolution is a larger image.

(c) Arbitrary zero-padding, with stride: the domain is enlarged and padded with zeroes, but not enough to capture the boundary pixels. Further, each discrete step skips one pixel. The result is the same as no stride followed by downsampling.

Figure 6.13: Boundary conditions and stride.

6.3 Convolutional Neural Networks

Now, we have all we need to introduce CNNs. On the surface, CNNs are just normal feed-forward neural networks in which in each layer, instead of the linear operation of matrix multiplication with the weights and biases, we have another linear operation: convolution.

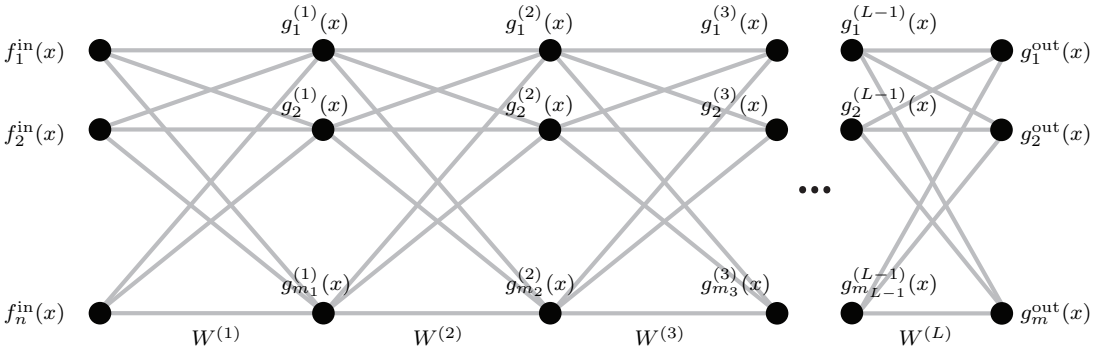


Figure 6.14: CNN consisting of L convolutional layers. Notice the architecture superficially is identical to a feed-forward neural network.

This modified version of the network layer (unsurprisingly) takes the name of *convolutional layer*. Such layers perform the following operation on the incoming data:

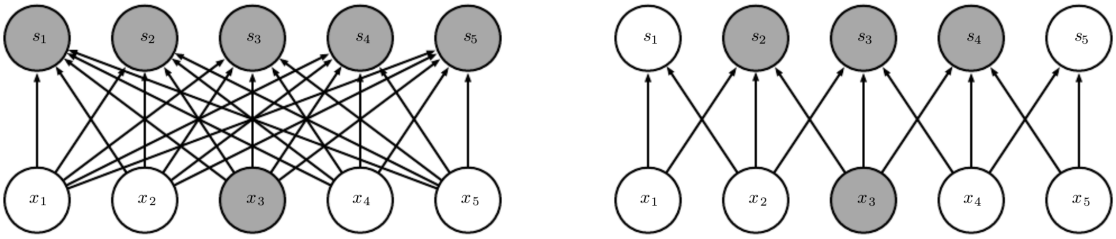
$$g_\ell^{(k)}(x) = \sigma\left(\sum_{\ell'=1}^{m_{k-1}} (g_{\ell'}^{(k-1)} \star w_{\ell,\ell'}^{(k)})(x)\right) \quad \begin{matrix} \ell = 1, \dots, m_k \\ \ell' = 1, \dots, m_{k-1} \end{matrix} \tag{6.21}$$

in which the activation function is left unchanged, but the parameters are no longer weight matrices but *filter* (or *kernel*) matrices. If the data under consideration are images, then the incoming $g_{\ell'}^{(k-1)}$ are feature maps computed by the earlier convolutional layer, that still have the shape of images (although maybe with different dimensions), and thus that can be again convoluted in the current layer.
Where is the advantage? Well for starters, we have seen how the convolutional operator by itself implements shift-equivariance, one of our desirable priors. Then, we have a huge gain in computational complexity, since the number of parameters per filter is *constant* with respect to the size of the input, while instead in the standard MLP a fully-connected layer has a weight matrix that must have dimensions matching the incoming and outcoming shape of the data.
This is possible since the *same filter* is applied to the whole data (across the whole image), and hence we have *weight sharing*. In fact, if we apply eq. (6.20) with the first term being the earlier feature map and the second being the filter, we have

$$(g_{\ell'}^{(k-1)} \star w_{\ell,\ell'}^{(k)})(x)[m,n] = \sum_i \sum_j g_{\ell'}^{(k-1)}[i,j]\, w_{\ell,\ell'}^{(k)}[m-i, n-j]. \tag{6.22}$$

in which i, j will span the filter (according to the boundary condition), eventually spanning all its weights, therefore the same weights will be used also to compute the other locations of the current feature map.
This fact leads to weight sharing and efficient computation (since the complexity of the sums above depend on the size of the filters, that do not depend on the size of the input), but also to

sparse interactions. In fact, not every location of the earlier feature map will be used to compute a single location of the current feature map, but only the ones "covered" by the filter.

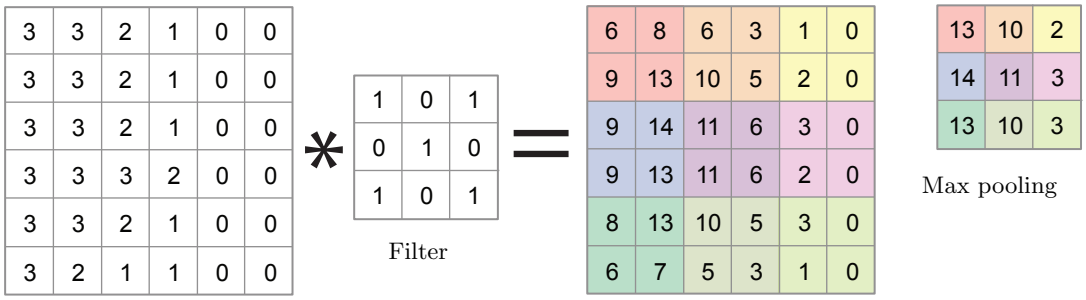


(a) Fully-connected layer: The outgoing edges from the input node x_3 have different weights than the ones outgoing from the other nodes.

(b) Convolutional layer: The outgoing edges from the input node x_3 have *the same* weights of the ones outgoing from the other nodes. Furthermore, notice that there are fewer edges in general, leading to *sparse interactions*.

Figure 6.15: Visualization of weight sharing.

This is not all however. The other main architectural difference is the introduction of another new operation: *pooling*. A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs (its *neighborhood*). The most used pooling functions are the *max pooling* that replaces the neighborhood with its maximum value and the *avg pooling* that instead does so with its average value.



Input data

Filter

Feature map

Max pooling

Figure 6.16: Pooling operation.

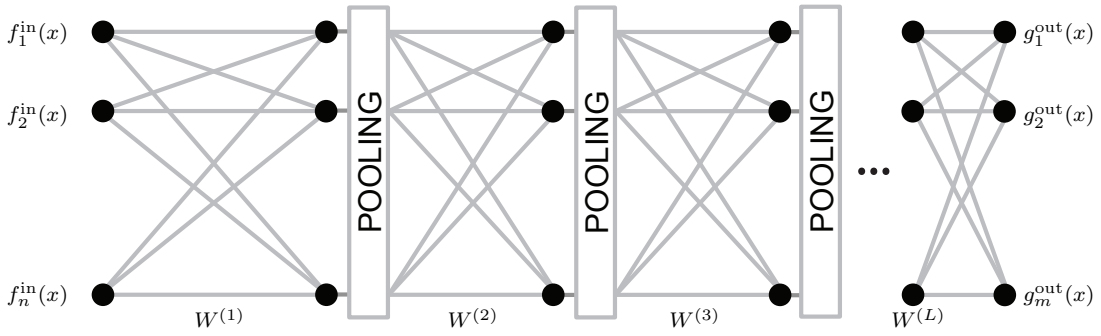


Figure 6.17: CNN consisting of L convolutional layers interleaved with pooling

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. This allows to capture non-local interactions via simple building blocks that only describe sparse interactions. Furthermore, if we pool over the outputs of separately parametrized convolutions (different layers), the features can learn which transformations to become invariant to.
Pooling also serves as a *subsampling* operator, reducing the size of the data and thus the computational and statistical burden on the next layer.

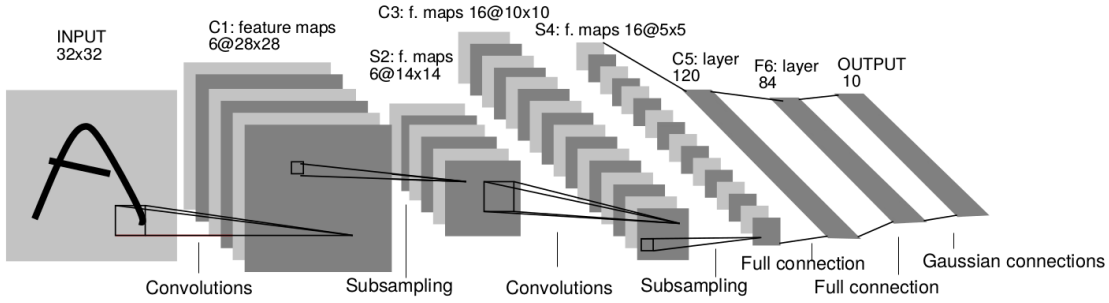


Figure 6.18: Convolution and downsampling (pooling) operations give the typical shape of a CNN, with data decreasing in size but filters increasing in number, then followed by fully-connected layers to perform other tasks like classification, based on the representation of the data the previous layers have extracted.

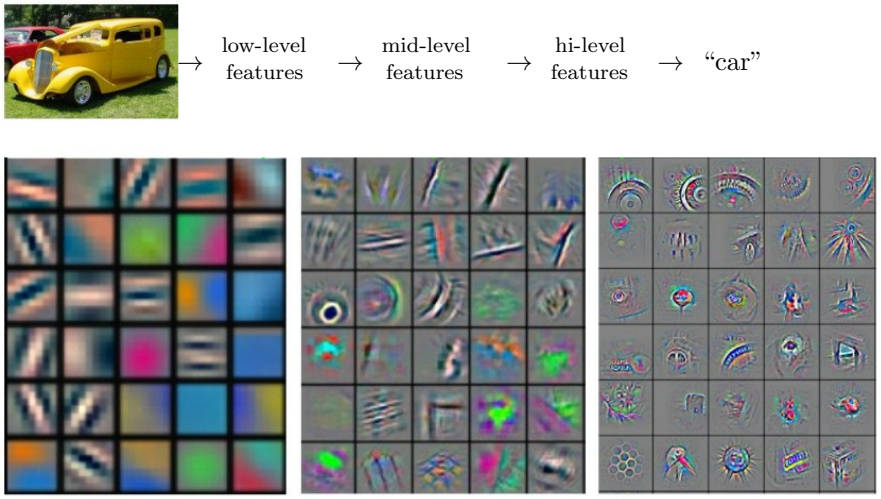


Figure 6.19: Empirically it has been shown that it is possible to interpret the CNN architecture as learning increasingly more complex features.

Chapter 7

Regularization

In this chapter we are going to see a few ways for *regularizing* the predictions produced by our networks.
A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the *generalization* error (i.e. test error), possibly at the expense of increased training error. These strategies are known collectively as *regularization*.
Regularization does not have a strict definition, since these strategies are very diverse in nature as we will see. A commonly accepted and loose definition for regularization is the following:

"Any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error".

Most of the research in Deep Learning is, in one way or another, doing regularization or providing new ways of regularizing deep networks.

Overfitting We have seen that overfitting is one of the main causes of a model losing generalization power, and that overfitting often happens with limited training data:

$$\#\text{parameters} \gg \#\text{training examples}. \tag{7.1}$$

Since usually we don't have enough data to be able to increase the number of training examples, one of the objectives of *regularization* is to reduce overfitting, in order to attain better *generalization*.

General idea One of the reasons of overfitting is the excessive *representational power* of the model wrt to the training data: the model is "too powerful" and is able to perfectly represent the training data, making implicit assumptions about the data distribution that are often false in the general setting, and so will lead to poor generalization on unseen data that does not follow these assumptions.
Therefore, many regularization techniques are aimed at reducing the number of *free parameters* (different from the number of weights) of a model, to limit its representational power. This is done by constraining the parameters of the model to behave in a specified way, limiting their freedom. This limitation may take several forms.

- **Eliminate** network weights.
  This can be done by taking a trained network, looking at it's parameters and estimating

the network sensitivity wrt to each individual weight, i.e. how much the output changes in response to a change in that weight. Parameters with little to no influence in the prediction can be eliminated.

- Weight **sharing** (i.e. # weights < # connections).
  Doing weight sharing means that we have less trainable weights. This is present in the convolutional layers, where each location of the current feature map is computed from the earlier feature map reusing the same kernel with the same weights.
- Explicit **penalties**. For instance we have seen Tikhonov regularization when we first encountered overfitting.
- **Implicit** regularization.

7.1 Explicit ways of regularization

We can add a *weight penalty* as a *regularizer* term to our *loss*, introducing a trade-off between *data fidelity* and *model complexity*.

$$\underbrace{\ell(\boldsymbol{\Theta})}_{\text{loss}} + \lambda \underbrace{\rho(\boldsymbol{\Theta})}_{\text{regularizer}} \tag{7.2}$$

When minimizing the loss function, the optimization algorithm (e.g. SGD) we are using will change the model parameters not only to improve its capacity of fitting the training data, but also to enforce some (soft) constraint induced by the regularizer.
Many regularizers enforce some form of *norm* penalty, trying to penalize the weights for growing too much. The optimizer will understand (from the gradient) that decreasing the very large parameters will lead to a large decrease in the loss function, even if at the expense of some fitting error (hence the tradeoff), and will change the parameters accordingly.
Beside offering a way to potentially reduce overfitting by limiting the expressivity of the model, this often offer also the possibility of *faster training*, since the regularizer will prevent the updates with a larger learning rate from becoming too large.

Typical penalties are:

- Tikhonov (L_2) regularization.

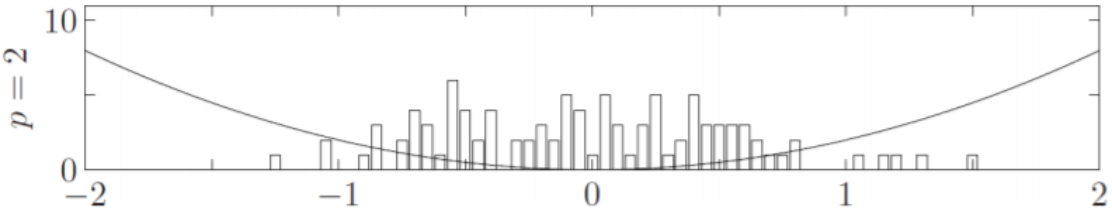


Figure 7.1: An histogram of the values taken by the weights shows how this regularizer tends to minimize an x^2 penalty for each weight, promoting *shrinkage*. Notice how the quadratic penalty translates to having higher penalty on weights with larger norms, and viceversa.

- Lasso (L_1) regularization.

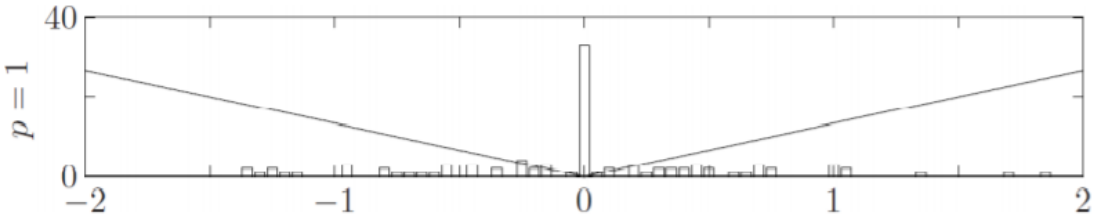


Figure 7.2: An histogram of the values taken by the weights shows how this regularizer tends to minimize an x penalty for each weight, promoting *sparsity* or *weight selection*. Notice how every weight receives (proportionally) an equal penalty, and so most of them are at zero.

- Bounded L_2 norm at each layer $\left\|\mathbf{W}^{(\ell)}\right\|_F \leq c^{(\ell)}$, meaning that instead of just promoting small L_2 norm, it places an upperbound c on the L_2 norm of the weights at a layer ℓ.

  Note. Formally we use the *Frobenius* norm, which generalizes the L_2 norm to matrices, being the square root of the sum of all their entries squared.

$$\|M\|_F = \sqrt{\sum_i \sum_j |M_{i,j}|^2}. \tag{7.3}$$

After training, the L_p magnitude of each weight reflects its importance in the final solution.

Sparsity Let's understand better how sparsity emerges with lasso regularization. We will use an example in $\mathbb{R}^2$, meaning we are considering a model with parameters $\boldsymbol{\theta} = \left(\theta_1, \theta_2\right)^\top$, just to better visualize the situation.
The loss function $\mathcal{L} : \mathbb{R}^2 \to \mathbb{R}$ can be visualized as isolines on the plane. Suppose $\boldsymbol{\theta}_{opt}$ is where the optimum (minimum) of $\mathcal{L}$ is, which would be the result of a simple minimization of the loss function. Then with a minimization algorithm we would iteratively shift the weights from an initial point eventually to this point. Now suppose that we don't just want to minimize $\mathcal{L}$, but we are also enforcing a (hard) constraint: we want

$$\|\boldsymbol{\theta}\|_p \leq 1; \tag{7.4}$$

then, in fig. 7.3 we can see this with $p = 1$ (L_1 penalty) and $p = 2$ (L_2 penalty).

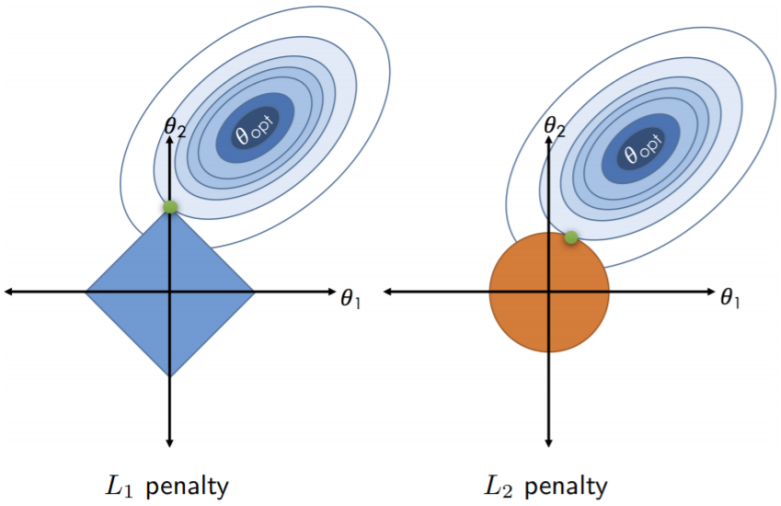


Figure 7.3: L_1 penalty vs L_2 penalty.

Since the constraint is a hard constraint, the point $\boldsymbol{\theta}$ is constrained to lie inside the locus of points satisfying eq. (7.4), and so to come as close as possible to the optimum, it will lie on the edge. Notice that, using the L_1 penalty, the θ_1 value goes to 0; this is what we would call sparsity. This is not by chance; in fact, it can be formally shown that selecting a point at the corners of the L_1 ball is much more likely to happen than selecting a point at the intersection between the sphere and an axis in the L_2 ball. For L_1 we will have very often a point on the corner (*sparse solution*); for the L_2 we will have very often a point on the arc (*non-sparse solution*).

7.2 Early stopping

Early stopping is a regularization technique that stops training as soon as performance on a validation set decreases.

U-shaped curve When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but *validation error* begins to rise again.

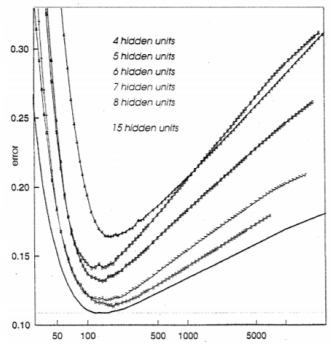


Figure 7.4: Plot of the validation error when increasing the training time showing a typical U-shape. Each curve corresponds to a different model size (increasing from top to bottom).

A U-shape plot shows how as the training goes on, the validation error decreases and then increases again, while the training error always decreases: when this happens it means that the model is overfitting on the training data. The plot suggests that there exists a moment in time at which the training should stop, because from that point on the validation error would increase, so the model is not getting better at the (general) task, but is in fact getting worse.
Things to keep in mind:

- Small networks can also overfit.
- Large networks have best performance *if they stop early.*

However, there is no fixed recipe.

- Different models, with different loss functions, on different tasks, with different datasets, would exhibit different curves. From now assume to use *SGD* to train a *MLP.*
- Since we have to take decision on whether to stop or not "on the fly", we cannot be certain that at the current moment the validation loss is definitely increasing, but might be just a temporary increase due to noise. For this reason it is useful in practice to define *patience* as the numers of training steps (or epochs) we allow the validation loss to increase. If after such interval of time the validation loss has not decreased by at least some amount $\Delta\ell$, we assert that it is indeed definitely increasing.

Overfitting It is important to note that it is not always true that more parameters lead to overfitting.

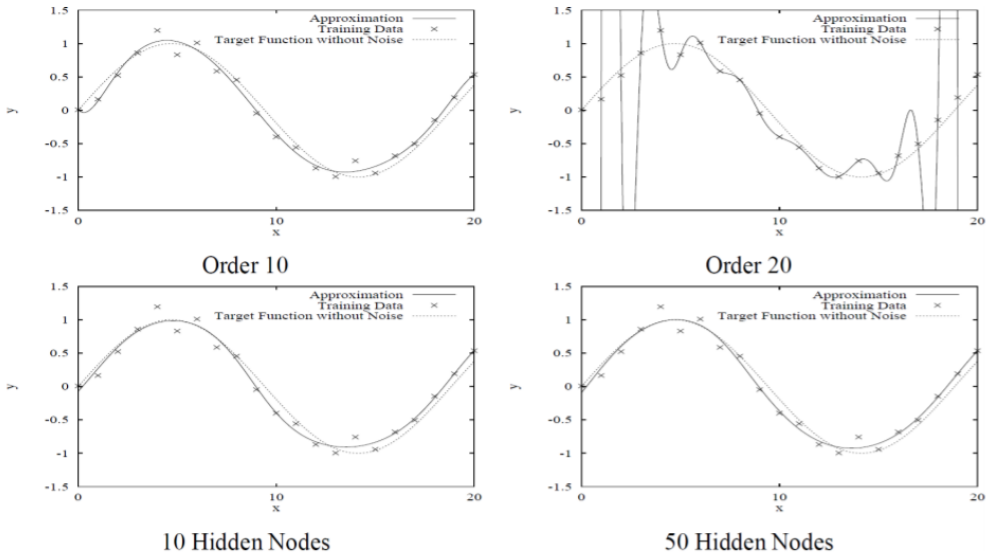


Figure 7.5: More MLP parameters *not* leading to overfitting.

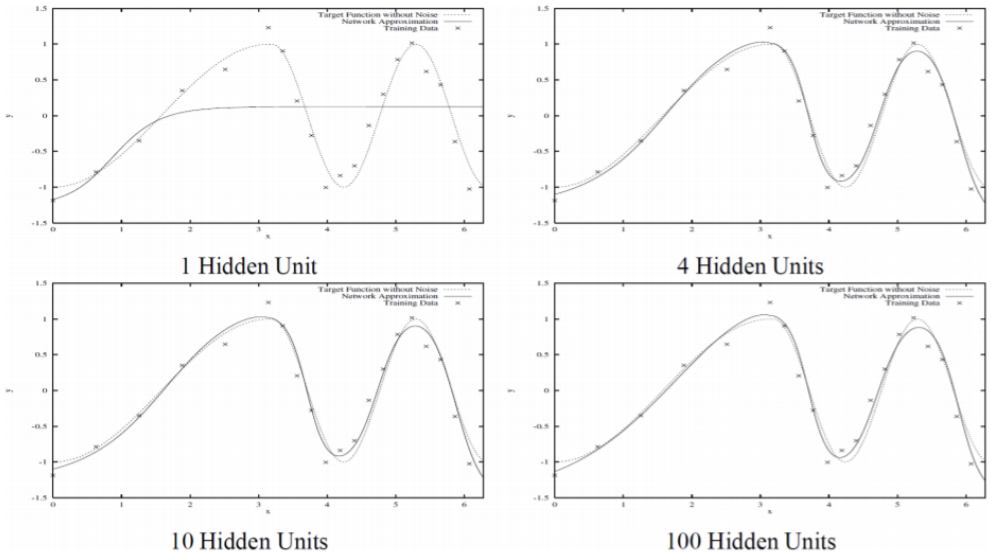


Figure 7.6: Good fit over all the different *data regions*.

Overfitting is *local* and can vary significantly in different regions:

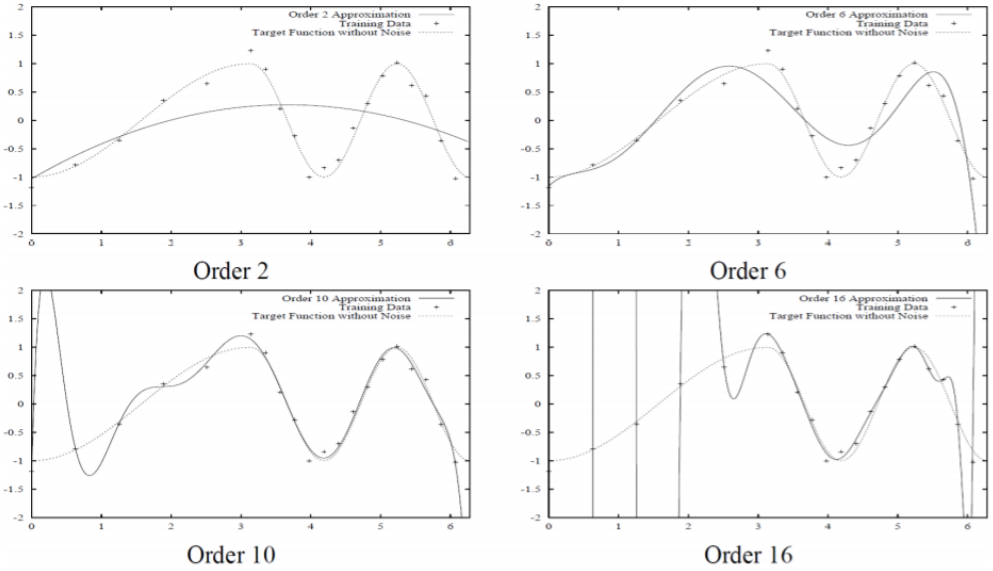
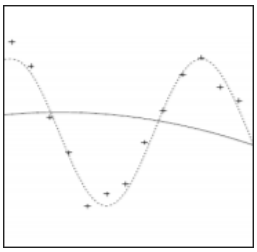
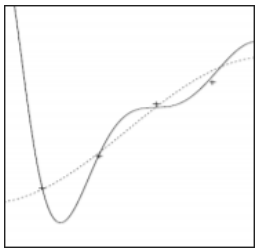


Figure 7.7: Example of overfitting and underfitting in localized *data regions*.



(a) Avoid underfitting regions of *high* nonlinearity.



(b) Avoid overfitting regions of *low* nonlinearity.

Figure 7.8: Two important requirements for *early stopping*.

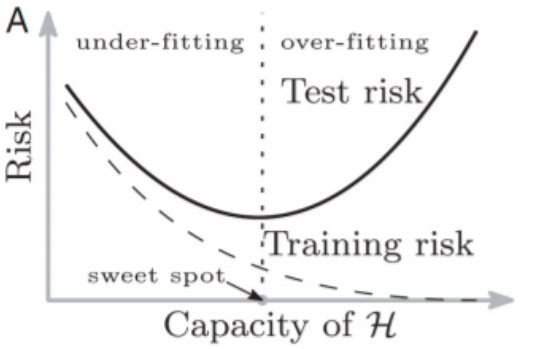


Figure 7.9: Plot of the training and validation loss as a function of model *capacity* $\mathcal{H}$.

Double descent (Capacity-wise double U-shape) Looking at fig. 7.9 we recognize the same U-shape seen before (*risk* can be used instead of *error*), but notice the subtle difference: before the plot of the validation error was done as a function of time, while instead this is as a function of *number of network parameters* called *capacity*. The fact that the two plots share similarly shaped curves empirically suggests that the number of parameters and the training time play a similar role on the validation error.
From the plot we can see how the training error is always decreasing as one increases the number of parameters in the network, since the network becomes more and more powerful, meaning it has a larger capacity, i.e. the set

$$\mathcal{H} = \{\text{functions that the network can represent well}\}. \tag{7.5}$$

is larger. Eventually, the capacity is so large that there exists a function that perfectly describes the data, so the training error goes to zero. We are overfitting.
The *sweet spot* represents the optimal capacity of the model, meaning it contains a function that describes both the training data and the validation data well enough.

Note. Early stopping does not directly influence the decision of the optimal capacity of a model, since it regularizes the training (so we have the training time on the x-axis in previous plots).

However, (very) recent results show a somewhat counterintuitive behavior.

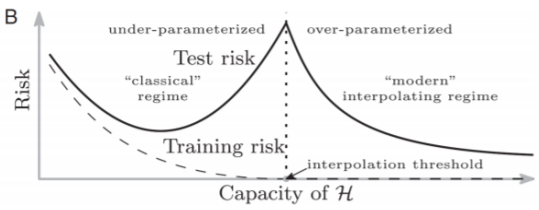


Figure 7.10: Plot of *double descent* curve.

In fig. 7.10 we see how as one keeps increasing the size of the network, it has been observed that the validation error will eventually start decreasing again, exhibiting what is referred to as *double*

descent curve or *double U-shaped* curve. The point where this happens is called *interpolation threshold*, and it is where we have a perfect fit on the training data (training loss is 0).
Although the capacity of $\mathcal{H}$ is already large enough for the training to bring the training error at 0, further increasing the size of the network and keeping training will not increase the training error, but nonetheless the weights will keep changing a little bit, but also decreasing the validation error.
This phenomenon has been observed but why it happens is not very clear yet. One explanation could be that by increasing the number of parameters, such that $\mathcal{H}$ will be a larger set of *function classes* and thus will contain more candidate functions, eventually $\mathcal{H}$ will become so large that one would find a function perfectly compatible with the training data, but that also fits the validation data perfectly.
Quoting the authors of the work that analyzed this behavior:

> "By considering *larger function classes*, which contain more candidate predictors compatible with the data, we are able to find interpolating functions that have *smaller norm* and are thus "*simpler*". Thus, increasing function class capacity improves the performance of classifiers."

What they mean by "interpolating functions that have *smaller norm* and are thus *simpler*", is that they are basing their assumptions on the *Occam's razor*. If one considers the smoothest (smaller norm) function that fits the data to be the simplest, then by Occam's razor we prefer simpler explanations to the data; however, one may encounter such a function only much later as one increases capacity.
This may seem obvious, since as the capacity of the model goes to infinity, eventually the model can represent any possible function. However, it is indeed surprising that SGD is able to find such good models, producing this double U-shape. As we have said, the training error after the interpolation threshold is nearly 0, so the gradients driving SGD will not be very informative; nevertheless in all that noise SGD is able, even quite consistently, to find these models.
Different optimization methods from *Stochastic Gradient Descent* (e.g conjugate gradient) yield worse generalization because they tend to overfit regions of low non-linearity.

Epoch-wise double U-shape It has been observed that the double U-shape of the validation loss also appears as a function of training time.

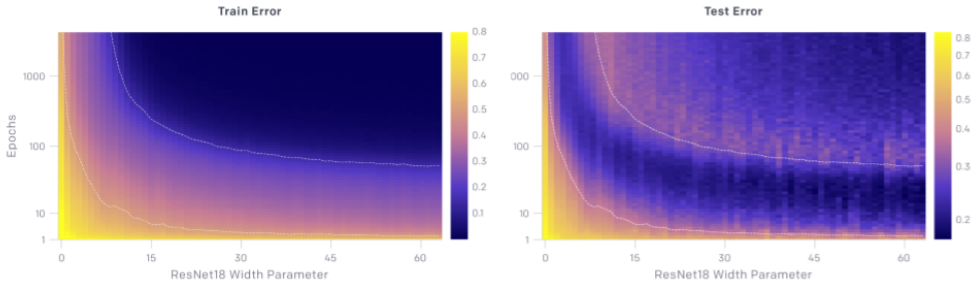


Figure 7.11: Train error and test error, plotted both against number of parameters and against training time.

In fig. 7.11 we can see how for each number of parameters, (taking a "verical slice" in the plot) the training error decreases with time (number of epochs). The same can be said if fixing the

number of epochs (taking a "horizontal slice in the plot") and considering it as a function of model capacity (width parameter). As we expect, the training error smoothly decreases as a function of both.
On the other hand, it has been observed that the test eror shows the double U-shape not only as a function of model capacity, as we have shown before, but also as a function of training time. This means that there is a regime where *training longer reverses overfitting.*

How early stopping acts as a regularizer So far we have stated that early stopping is a regularization strategy, but we have supported this claim only by showing learning curves where the validation set error has a U-shaped curve. What is the actual mechanism by which early stopping regularizes the model?
Early stopping is based on the *smoothness* heuristic:

Representational power *grows* with weight magnitude (training time).

This implies the following behavior during training:

- Initialize the net with small weights $\boldsymbol{\theta}_0$.
- As training time increases, simple hypotheses are considered before complex hypotheses.
- Training first explores models similar to what a *a smaller net of optimal size* would have learned.

More formally, imagine taking τ optimization steps (corresponding to τ training iterations) and with learning rate α. We can view the product $\alpha\tau$ as a measure of effective capacity: the model can represent all the functions that can be parametrized by parameters in a sphere of volume proportional to $\alpha\tau$ in the parameter space.
It has been argued that early stopping has the effect of restricting the optimization procedure to a relatively small volume of parameter space in the neighborhood of the initial parameter value $\boldsymbol{\theta}_0$. Therefore, assuming the gradient is bounded, restricting both the number of iterations τ and the learning rate α limits the volume of parameter space reachable from $\boldsymbol{\theta}_0$.
In this sense, $\alpha\tau$ behaves as if it were the reciprocal of the coefficient used for weight decay. Indeed, it can be shown how – in the case of a simple linear model with a quadratic error function and simple gradient descent – early stopping is equivalent to L_2 regularization.
This heuristic seems to reflect quite well what happens in practice, and thus early stopping is very easy to implement: stop learning when the validation error increases again.

7.3 Batch normalization

Batch normalization is a method of *adaptive reparametrization*, motivated by the difficulty of training very deep models. Very deep models involve the composition of several functions or layers. The gradient tells how to update each parameter, under the assumption that the other layers do not change. In practice, we update all of the layers simultaneously.
Consider a layer of a multi-layer perceptron (MLP):

$$\mathbf{x}^{(k)} = \sigma\left(\mathbf{W}^{(k)}\mathbf{x}^{(k-1)}\right). \tag{7.6}$$

At each layer the distribution of the input that enters that layer changes all the time during training, since the weights $\mathbf{W}$, and in particular the weights $\mathbf{W}^{(1:k-1)}$ of the earlier layers change

continuously, because it is what we are optimizing over. This phenomenon is called *internal covariate shift*.
It becomes a problem because the layers need to continuously adapt to the new distribution. This leads to slower training of the network than it could be if the input distribution did not change, since in that case the network would not have to account for this change.
Batch normalization is designed to try and *fix* the input distribution at each layer. This is done by *normalizing* the input features at a layer k by the statistics (*mean* and *variance*) computed on the entire training set after it has passed through the network and reached the k-th layer.

$$\hat{\mathbf{x}} = \text{normalize}(\mathbf{x}, \mathcal{X}) \tag{7.7}$$

where both $\mathbf{x}$ and $\mathcal{X}$ (all the training set) are parametrized by $\mathbf{W}$ because both of them have passed through the network until the layer k.
So the transformation $normalize(\cdot, \mathcal{X})$ is another transformation that the network must do, therefore to be able to train the network via a gradient descent-like algorithm we will need backprop to be able to propagate gradients through it, i.e. we need the transformation to be *differentiable*, to admit partial derivatives

$$\frac{\partial}{\partial \mathbf{x}}\text{normalize}(\mathbf{x}, \mathcal{X})\,, \qquad \frac{\partial}{\partial \mathcal{X}}\text{normalize}(\mathbf{x}, \mathcal{X}) \tag{7.8}$$

needed for the chain rule.

The transformation Let's see in detail what $normalize(\cdot, \mathcal{X})$ looks like.
In the univariable case, normalizing an input variable (also called *standardization* or *whitening*) would mean transforming this variable in order to obtain a mean of 0 and a variance of 1. This is done by removing the *mean* over the training set and dividing by the *standard deviation* over the training set:

$$\mathbf{x} \mapsto \frac{\mathbf{x} - \mathbb{E}_{\mathbf{x}\in\mathcal{X}}[\mathbf{x}]}{\sqrt{\text{var}_{\mathbf{x}\in\mathcal{X}}[\mathbf{x}]}}. \tag{7.9}$$

Learnable parameters: Scale and shift The transformation outlined so far could change the way the network was trying to operate, limiting its capacity.
Imagine that the optimal transformation that the network is trying to learn at a certain layer is the *identity*; using *batch normalization* this is no longer possible unless the variance is already 1 and the mean is 0.
To address this problem we add some new *trainable* weights that allows the net to learn the identity transformation (allowing the net to learn the identity implies that it will be able to learn everything).

$$x_i \mapsto \gamma_i \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\text{var}(x_i)}} + \beta_i \tag{7.10}$$

These allow to represent the identity $x_i \mapsto x_i$, and so to *undo* the normalization, if that was the optimal thing to do. This preserves the *representational power* of the original network.

Using mini-batches Doing this kind of normalization wrt the *entire* training dataset is going to be very costly. Instead, the idea is to use *mini-batches*. At each layer the *mean* and the *variance* are estimated only wrt to the current *mini-batch* at each parameter update.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 7.12: Statistics calculation on mini-batches.

The batchnorm transformation makes each training example *interact* with the *other examples* in each mini-batch, transforming each data point wrt to the other data points in the mini-batch.

Properties Typically, batchnorm is applied right before the nonlinearity:

$$\sigma(\mathbf{Wx} + \mathbf{b}) \quad \text{becomes} \quad \sigma \circ \mathrm{BN}_{\gamma,\beta}(\mathbf{Wx}) \tag{7.11}$$

The *bias* can be removed, since when doing the mean subtraction it undoes any shift that has been applied. If the shift was needed to have an optimal solution to the network, the new shift is the β_i.
At *test* time it is not possibile to transform a data point wrt to the *mean* and *variance* of the *mini-batch*, because mini-batches are not legitimate. During training a batchnorm layer estimates the mean and variance of the training set and stores them. Since the statistics of different minibatches will likely be different, some king of moving average is kept. At test time the mean and variance used for the normalization of a data point are these averages of the statistics estimated during training.
Beneficial properties include:

- Since in each epoch a sample may end up in different minibatches, each time the sample will be transformed in a slightly different way, depending on the mean and variance of the mini-batch that contains it (so depending on the *interaction* with the *other examples* in the mini-batch). Therefore, the stochastic uncertainty of the batch statistics acts as a *regularizer* that can *benefit generalization*: the network tends to become more robust to variations.
- Batchnorm leads to more *stable gradients*, meaning that one can increase the learning rate and thus achieve *faster training* with more safety.

Variants Some tried to bring variations to the normalization idea, since it has been argued that normalizing along the *batch dimension* can lead to inconsistency:

- Bad transfer across different data distributions. If the training has been made on a certain data distribution, then the network won't transfer well on completely new data.
- The stats must be reliable. The size of the mini-batch determines the quality of the mean and variance estimates; reducing the *mini-batch size* increases the model error dramatically.

For this reasons, several variants have been proposed:

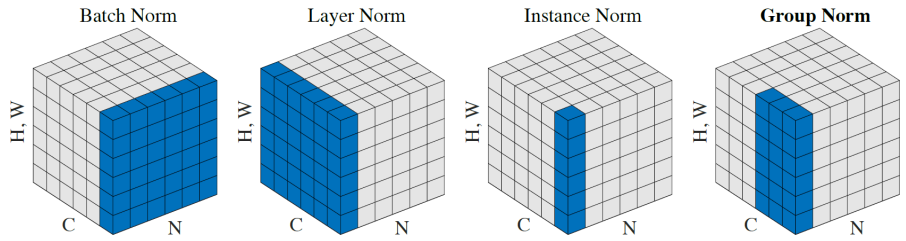


Figure 7.13: Visualization of the action of batchnorm and its variants on the incoming batches of matrices.

Looking at fig. 7.13, suppose we have a minibatch of N samples, with each sample being a tensor of dimensions (H, W, C) (e.g. an RGB image, with the spatial dimensions H, W flattened). Then:

- **Batchnorm**: Normalizing each channel independently along the direction of the batch.
- **Layer norm**: Normalizing each sample independently along the direction of the channels.
- **Instance norm**: Normalizing each channel of each sample independently.
- **Group norm**: Normalizing across the channels of each sample, but within some radius.

7.4 Dropout

Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making bagging practical for *ensembles* of very many large neural networks.
Assume you have unlimited computational power. It would be possible to generate thousands of deep networks, train all of them on the same data and then take the average prediction as final prediction. In machine learning this way of proceeding is called *ensemble* machine learning. Ensemble predictions (e.g. bayesian networks, random forests) are known to generalize better than the individual models.
Given a multi-layer perceptron network, we could generate an ensemble of deep neural networks by randomly removing some of the nodes. This will result in a different network and therefore will represent a different function. Dropout can be considered an ensemble method for deep learning, which parametrizes each model in the ensemble by *dropping* random units (i.e. nodes with their input/output connections) of the "main" network.

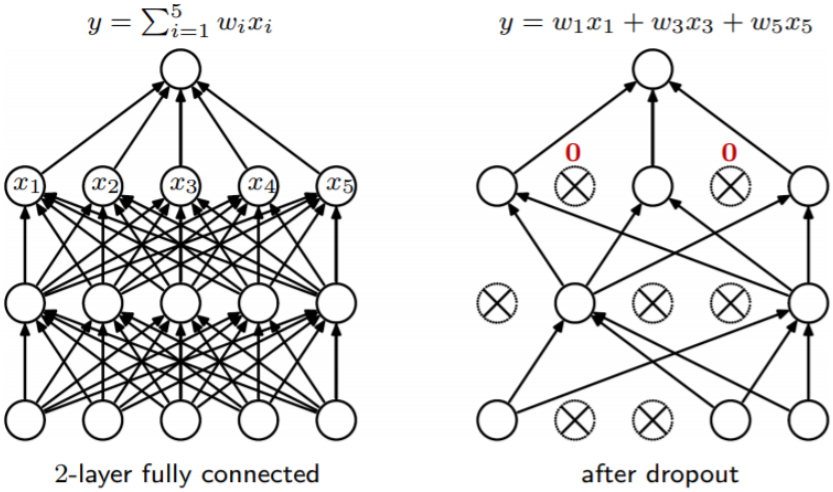


Figure 7.14: Before and after dropout.

Implicit ensembles Dropout has two distinguishing features:

- It does *bagging*: for a family of models, each model is trained on a subset of the data (e.g. mini-batches).
- It does *weight sharing*, which is *atypical* in ensemble methods.

In fact, since we do not have unlimited computational power and thus it would be infeasible to *explicitly* generate an ensemble of different configurations, dropout *implicitly* defines the ensemble keeping just *one single network* and randomly removing (zero-out) some of its units, each with a specified probability p of being retained, chosen by the user, globally for the whole network or also *per layer*.

Note. How large would an explicit ensemble built from sampling the network be? If the network contains n units, then each sampled network can either be present in the network or not, so we have 2^n possible configurations.

Training At training time, dropout generates a new sampling of the network each time new training data is presented (e.g. *at each mini-batch* in SGD), that thus enters a completely new configuration of the network.
The individual models (samples of the original network) are *not* optimized to convergence because they are "discarded" in favor of new, different models at each optimization step.
However, the *ensemble* is trained to convergence: at each training step the weight update is applied to all members of the ensemble (i.e. the original network) simultaneously.

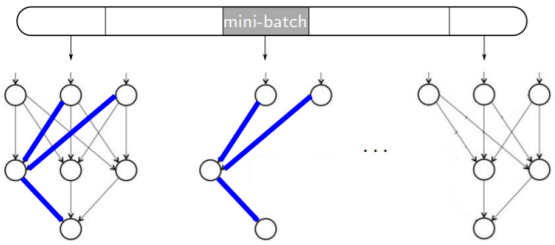


Figure 7.15: New models are generated one by one as one produces new mini-batches during optimization. Notice the weight sharing, e.g. the blue edges have the same weights in every model that contains them.

Testing At test time ensembles typically produce their output as some form of *average* of the output of all their members. Since in the case of dropout the members are defined only implicitly, the trained weights from each model in the ensemble must be *averaged* somehow. The simple idea is that if a unit is retained with probability p during training, its outgoing weights are multiplied by p, since the average contribution of that unit in training the ensemble has been p.

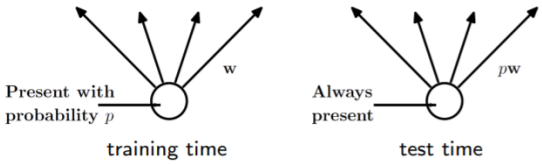


Figure 7.16: Dropout: training time vs test time.

Properties as a regularizer In a standard neural network, weights are optimized *jointly*. This means that errors in the optimization are spread across all the weights. If a unit in the neural network is making mistakes (e.g. has learnt a sub-optimal hidden representation) then probably there will be some later unit in the network making up for it. Units are allowed to do mistakes because these mistakes will be absorbed by some other unit. This is called *co-adaptation* in the literature.
Although it may seem like a nice failsafe, it is actually something we want to avoid because it worsen the *transferabilty* of the network to unseen new data.

- Dropout *reduces* co-adaptation by making units unreliable (they can appear and disappear suddenly, so earlier units that relied on later units to correct their mistakes are discouraged to do so). This improves *generalization* to unseen data, and reduces overfitting.
- Dropout learns *sparse* representations, meaning that the intermediate variables will tend to be sparse. This is a nice side-effect that has been observed in practice, since dropout is not explicitly designed to do so.
- Performs closely to *exact* model averaging over all 2^n models and much better if no weight sharing is done in the exact model.
- *Longer* training times (usually 2 to 3 times longer), since parameter updates are now noisier. Dropout is another source of stochasticity in addition to the mini-batches.
- Typical choices: 20% of the input units and 50% of the hidden units.

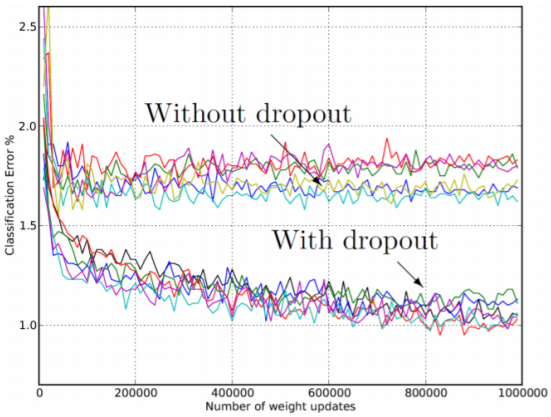


Figure 7.17: Validation error comparison.

Dropout as an explicit penalty It has been shown that in a 2-layer network

$$\hat{\mathbf{y}} = \sigma(\mathbf{V}(\sigma(\mathbf{U}\mathbf{x}))) \tag{7.12}$$

applying dropout is formally equivalent to changing the loss function by including the following weight penalty:

$$\ell(\mathbf{U}, \mathbf{V}) + \frac{1-p}{p} \sum_{i=1}^{r} \|\mathbf{u}_i\|^2 \|\mathbf{v}_i\|^2 \tag{7.13}$$

where the vectors are the columns of $\mathbf{U}$ and $\mathbf{V}$. This is a form of *path regularizer*, since it computes a sum of products of squared weights along each possible path from input to output. This regularizer term tends, at the optimum, to equalize the norms $\|\mathbf{u}_i\|^2, \|\mathbf{v}_i\|^2$ for all i, which is conjectured to reduce co-adaptation since it is equalizing the contribution of each unit. For a small enough dropout rate, it can be shown that all minima are *global*.
In practice this is not very useful to us, but it is interesting to appreciate how an implicit regularizer such as dropout can be turned into an explicit penalty expression.

Chapter 8

Deep generative models

A *generative* model is a statistical model of a *distribution* of some data, namely of the *probabilistic process* that produced the data.
Generative models learn a distribution from some given training samples and therefore are able to *generate* new samples from the learnt distribution. The quality of the generation will depend on how well the learnt distribution approximates the real one.
Deep generative models are the combination of generative models and deep neural networks.

Dimensionality reduction *Dimensionality reduction* is the name for a class of techniques and models that transform data from a high-dimensional space into a low-dimensional space, so that the low-dimensional representation (hopefully) retains most of the meaningful properties of the original data.
For this to be possible, a model must learn how to *generate* data from a lower-dimensional distribution that is as close as possible to the input higher-dimensional distribution; therefore, it can be regarded as a generative model.
Formally the task of dimensionality reduction is defined as follows.
Given n datapoints stored as columns of a matrix $\mathbf{X} \in \mathbb{R}^{d \times n}$ we want a similar representation of the matrix $\mathbf{X}$ with smaller dimension:

$$\mathbf{X}^\top = \begin{pmatrix} - & \mathbf{x}_1^\top & - \\ & \vdots & \\ - & \mathbf{x}_n^\top & - \end{pmatrix} \approx \begin{pmatrix} - & \tilde{\mathbf{x}}_1^\top & - \\ & \vdots & \\ - & \tilde{\mathbf{x}}_n^\top & - \end{pmatrix} = \tilde{\mathbf{X}}^\top \tag{8.1}$$

where $\tilde{\mathbf{X}} \in \mathbb{R}^{k \times n}$ with $k \ll d$.
Dimensionality reduction finds many uses, such as:

- Visualization: visualizing higher dimensional data (e.g. 3D, with $d = 3$) in the plane ($k = 2$).
- Denoising and outlier detection: these can be seen as useless additional information.

8.1 Principal component analysis

Principal component analysis is a technique of (linear) dimensionality reduction.
Given some points in a d-dimensional space, we are interested in identifying the direction(s) where data changes the most, to neglect the ones where data changes the least. The motivation

is, if most datapoints from a distribution share very similar coordinates along a direction (are very close), then neglecting this direction when representing the distribution should not result in a great loss of information content.
In particular, we want to find the $k \leq d$ *orthogonal* directions with the most *variance*. These will form the *basis* (hence the reason we prefer the set of directions to be orthogonal) of a k-dimensional subspace of the original d-dimensional space of the data, in which we will *project* the datapoints.
Example. In the plane we have $d = 2$, and we might be interested in the single principal component $k = 1$, and that would be the only dimension later used to represent the data.

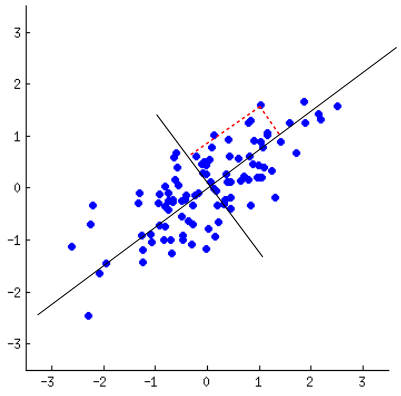


Figure 8.1: An example of PCA.

But what do we mean by projection? Suppose we have identified the k principal directions (we will see how to do that later), how do we express a datapoint $\mathbf{x}$ in terms of these directions? For each direction, the component of $\mathbf{x}$ wrt to that direction will be the distance from the origin of the *projection* of the point on the direction. This means that to compute the transformed datapoint $\mathbf{z}$ we have to compute k dot products between $\mathbf{x}$ and the k-th principal component vector $\mathbf{w}_k$ (a d-dimensional unit vector). This means that this transformation can be encoded by a matrix:

$$\mathbf{Z}^\top = \underbrace{\begin{pmatrix} - & \mathbf{z}_1^\top & - \\ & \vdots & \\ - & \mathbf{z}_n^\top & - \end{pmatrix}}_{n \times k} = \underbrace{\begin{pmatrix} - & \mathbf{x}_1^\top & - \\ & \vdots & \\ - & \mathbf{x}_n^\top & - \end{pmatrix}}_{n \times d} \underbrace{\begin{pmatrix} | & & | \\ \mathbf{w}_1 & \cdots & \mathbf{w}_k \\ | & & | \end{pmatrix}}_{d \times k} = \mathbf{X}^\top \mathbf{W}. \tag{8.2}$$

Since PCA assumes that the principal components are orthogonal, if $k = d$ we will have an *orthogonal matrix*:

$$\mathbf{W}^\top \mathbf{W} = \mathbf{I}_{k \times k} = \mathbf{I}_{d \times d} = \mathbf{W}\mathbf{W}^\top. \tag{8.3}$$

(see Claim A.1.)

This implies that for $k = d$ we have

$$\mathbf{X}^\top \mathbf{W} = \mathbf{Z}^\top \tag{8.4}$$
$$\mathbf{X}^\top \mathbf{W}\mathbf{W}^\top = \mathbf{Z}^\top \mathbf{W}^\top \tag{8.5}$$
$$\mathbf{X}^\top \mathbf{I}_{d\times d} = \mathbf{Z}^\top \mathbf{W}^\top \tag{8.6}$$
$$(\mathbf{X}^\top)^\top = (\mathbf{Z}^\top \mathbf{W}^\top)^\top \tag{8.7}$$
$$\mathbf{X} = \mathbf{W}\mathbf{Z}. \tag{8.8}$$

However, we want $k < d$, but in this case the matrix $\mathbf{W}$ will be only *semi-orthogonal*, since it is non-square. This means that it is

$$\mathbf{W}^\top \mathbf{W} = \mathbf{I}_{k\times k}, \qquad \mathbf{W}\mathbf{W}^\top \neq \mathbf{I}_{d\times d}, \tag{8.9}$$

and hence

$$\begin{aligned} \mathbf{X}^\top \mathbf{W} &= \mathbf{Z}^\top \\ \mathbf{X}^\top \underbrace{\mathbf{W}\mathbf{W}^\top}_{\neq \mathbf{I}_{d\times d}} &= \mathbf{Z}^\top \mathbf{W}^\top \\ &\vdots \\ \mathbf{X} &\approx \mathbf{W}\mathbf{Z}. \end{aligned} \tag{8.10}$$

So in the general case we have an exact *projection* transformation and an *approximate* reconstruction transformation

$$\mathbf{X}^\top \mathbf{W} = \mathbf{Z}^\top \tag{8.11}$$
$$\mathbf{X} \approx \mathbf{W}\mathbf{Z} \tag{8.12}$$

and from the lower representation $\mathbf{Z}$ recovering the full rank representation of the data is something that we cannot do in general.
We have seen that once the directions are set, the transformation is completely determined, so we have to choose an *optimal* set of principal components, wrt to some *criterion*. We want the projection in the lower-dimensional space to be as *lossless* as possible, or dually we want the reconstruction to be as accurate as possible.
So, our criterion to choose the set of principal components will be to minimize the reconstruction error (also called *projection error*)

$$\epsilon = \|\mathbf{X} - \mathbf{W}\mathbf{Z}\|_2^2 = \left\|\mathbf{X} - \mathbf{W}\mathbf{W}^\top \mathbf{X}\right\|_2^2, \tag{8.13}$$

which as it turns out is equivalent to maximizing the variance of the projected data, that recall was the requirement we started with.

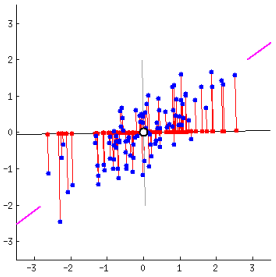


Figure 8.2: The direction along which the projection error is minimized is also the one over which variance is maximized.

Let's consider only one principal component $\mathbf{w}$ and, to simplify the calculations, let's assume that the data points $\mathbf{X}$ have zero mean, meaning that they are *centered* at zero. The projections of all n datapoints onto $\mathbf{w}$ is $\mathbf{X}^\top\mathbf{w}$. Now, reminding that the projection error is given by

$$\sum_i \|\mathbf{x}_i - \left(\mathbf{x}_i^\top \mathbf{w}\right)\mathbf{w}\|^2 = \sum_i (\mathbf{x}_i - \left(\mathbf{x}_i^\top \mathbf{w}\right)\mathbf{w})^\top (\mathbf{x}_i - \left(\mathbf{x}_i^\top \mathbf{w}\right)\mathbf{w}) = \sum_i \|\mathbf{x}_i\|_2^2 - 2(\mathbf{x}_i^\top \mathbf{w})^2 + (\mathbf{x}_i^\top \mathbf{w})^2\|\mathbf{w}\|_2^2 \tag{8.14}$$

Now, as $\mathbf{w}$ is orthonormal, $\|\mathbf{w}\| = 1$ and thus we are left with

$$\sum_i \|\mathbf{x}_i\|_2^2 - 2(\mathbf{x}_i^\top \mathbf{w})^2 + (\mathbf{x}_i^\top \mathbf{w})^2 \tag{8.15}$$

moreover, since we are maximizing wrt to $\mathbf{w}$, we can ignore the $\mathbf{x}_i$s and just maximize the following expression

$$-\sum_i (\mathbf{x}_i^\top \mathbf{w})^2 = -\left\|\mathbf{X}^\top \mathbf{w}\right\|_2^2 \tag{8.16}$$

or, equivalently, minimize its additive inverse

$$\left\|\mathbf{X}^\top \mathbf{w}\right\|_2^2 = \left(\mathbf{X}^\top \mathbf{w}\right)^\top \left(\mathbf{X}^\top \mathbf{w}\right) = \mathbf{w}^\top \underbrace{\left(\mathbf{X}\mathbf{X}^\top\right)}_{\mathbf{C}} \mathbf{w} \tag{8.17}$$

where $\mathbf{C} \in \mathbb{R}^{d\times d}$ is the symmetric *covariance matrix* of the data.
So we are looking for the vector $\mathbf{w}$ that minimizes the previous quadratic form, and since we are just interested in the direction we can consider the vector $\mathbf{w}$ as having unit norm. This leads to a *constrained optimization problem*:

$$\min_{\mathbf{w}} \mathbf{w}^\top \mathbf{C}\mathbf{w} \quad \text{s.t. } \|\mathbf{w}\|_2 = 1. \tag{8.18}$$

Constrained optimization problems can generally be solved by means of *Lagrange multipliers*, but for this specific type of problems (minimizing a quadratic form with $\mathbf{C}$ symmetric) we have a theorem that gives us an immediate solution. The *Courant minmax principle* tells us that the problem is solved *globally* (remember quadratic functions are convex so we have global optimality guarantees) by the first eigenvector of $\mathbf{C}$, which will be the principal component, and the corresponding eigenvalue is equal to the quadratic form.

If we want to find the *second* principal component $\mathbf{w}_2$, after having solved for the first one:

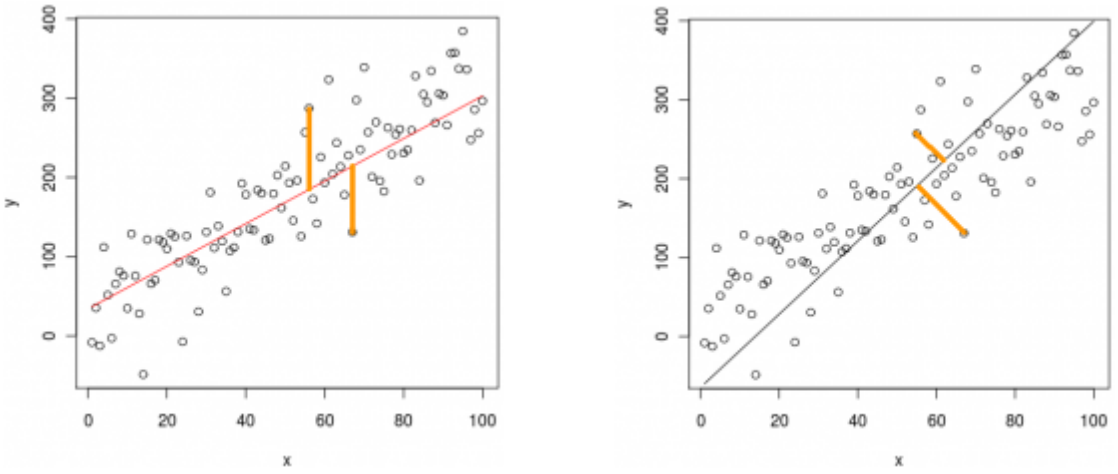
$$\mathbf{w}_1 = \underset{\mathbf{w}}{\operatorname{argmin}}\, \mathbf{w}^\top \mathbf{C}\mathbf{w} \quad \text{s.t. } \|\mathbf{w}\|_2 = 1 \tag{8.19}$$

we define a new constrained optimization problem, identical to the previous one but with the additional constraint enforcing the orthogonality of $\mathbf{w}_2$ wrt to $\mathbf{w}_1$:

$$\begin{aligned} \mathbf{w}_2 &= \underset{\mathbf{w}}{\operatorname{argmin}}\, \mathbf{w}^\top \mathbf{C}\mathbf{w} \\ \text{s.t.} &\begin{cases} \|\mathbf{w}\|_2 = 1 \\ \mathbf{w}_1^\top \mathbf{w} = 0 \end{cases} . \end{aligned} \tag{8.20}$$

Again, the Courant minmax theorem comes in handy, since it states the solution to this problem is the *second* eigenvector of $\mathbf{C}$, where the eigenvectors of $\mathbf{C}$ are ordered by increasing eigenvalues. If we iterate this procedure we can find the k principal components, that are thus the first $k \ll d$ eigenvectors of $\mathbf{C}$.

Note. PCA is not linear regression, since with the former we compute the error orthogonally to the principal direction while with the latter we compute the error along each coordinate, as can be seen in the following figure:



(a) Linear regression (b) PCA

Figure 8.3: PCA vs Linear regression

PCA as a generative model PCA can be seen as a generative model. Given the principal components $\mathbf{W}$ we can generate new data just by sampling $\mathbf{z}_{new} \in \mathbb{R}^k$ and computing the new sample using reconstruction eq. (8.12):

$$\mathbf{x}_{new} = \mathbf{W}\mathbf{z}_{new} \tag{8.21}$$

From a different perspective PCA gives us a parametric model. Each data point $\mathbf{x}$ is transformed into a low-dimensional *code* $\mathbf{z} \in \mathbb{R}^k$ where the dimension $k < d$ is fixed. The previous relations can be considered as encoding and decoding operations:

$$\mathbf{x}^\top \mathbf{W} = \mathbf{z}^\top \quad \text{(encoding)}$$

$$\mathbf{x} \approx \mathbf{W}\mathbf{z} \quad \text{(decoding)}$$

So passing from the code to the data point is a linear map, specifically a parametric map in which the parameters are the values of matrix $\mathbf{W}$. For this reason PCA is considered the simplest parametric model since the operation of encoding and decoding are both linear, and we can see this as linear generative model.
A more powerful but complex approach would be to model the encoding and decoding steps as deep neural networks, which grant more control and can be used to enforce the generation of meaningful data, avoiding the generation of gibberish. These, as we know, employ nonlinear transformations instead of linear ones.

8.2 Autoencoders

Autoencoders replace the linear encoding step and the linear decoding step with two deep neural networks, that by the universal approximation theorem are (theoretically) able to approximate any function, thus generalizing the idea of PCA as a parametric model.

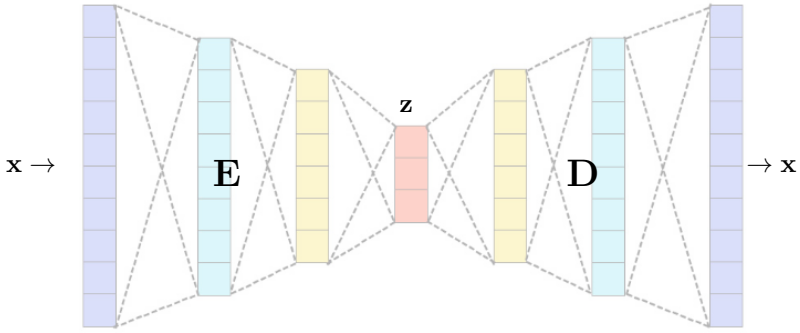


Figure 8.4: Autoencoder architecture.

Notice how the architecture of autoencoders has a *bottleneck* at the middle. The *encoder* function $E(\mathbf{x})$ will produce a *code* $\mathbf{z}$ that is lower dimensional, and that will be the input of the *decoder* function $D(\mathbf{z})$. The bottleneck is explicitly designed to make autoencoders unable to learn to copy perfectly, i.e.

$$D(E(\mathbf{x})) = \mathbf{x}. \tag{8.22}$$

In fact, we are not really interested in the final output of the autoencoder, but in the middle representation $\mathbf{z}$, also called *latent code* for $\mathbf{x}$. Since the model is able to copy only approximately, it is forced to prioritize which aspects of the input should be copied. This means that autoencoders often learn useful properties of the data.
The autoencoder is trained by minimizing the *reconstruction loss*:

$$\ell_\Theta = \sum_i \|\mathbf{x}_i - D_\Theta\left(E_\Theta\left(\mathbf{x}_i\right)\right)\| \tag{8.23}$$

where the choice of the specific metric depends on the data and on the task we are dealing with.

Note. If the layers of both the encoder and the decoder are linear, then the codes $\mathbf{z}_i$ span exactly the same space as PCA.

8.2.1 Manifolds

Manifold hypothesis How can such a mapping produce samples that resemble the observed data? It must be that there is some lower-dimensional structure (*manifold*) on which the data are constrained, and this structure is *embedded* in the higher-dimensional space.
Imagine text on a piece of paper: it naturally lives in the plane. However, if the paper is folded, we do not have a plane any longer, so we must represent this text in the 3D space. Nonetheless the text cannot leave the paper, it is constrained to lie onto it, but the paper (structure) is now *curved* and must be described (*embedded*) in a higher-dimensional space.
The decoder performs a mapping from a low-dimensional *latent space* to a high-dimensional *embedding space*. When we train an autoencoder we are learning a parametric model of a latent space, i.e. the underlying, lower-dimensional structure of the data.
We have already seen this concept: the *manifold hypothesis*. Given some data, e.g. images of the Eiffel tower, we can represent the data as set of points $\mathbf{x}$ in some high-dimensional space; however, the manifold hypothesis states that the way they are positioned in this high-dimensional space will not be random, but will have a *structure*, forming a "hyper-surface", or more precisely, a *manifold*. This structure encodes some information that all the datapoints have in common, therefore not every component of the points $\mathbf{x}$ are independent, but instead there are constraints that ensure that the points stay on the structure. The datapoints have *fewer degrees of freedom* than the dimensionality of the space.
If we know the *geometry* of such structure, i.e. if we know these constraints, we can drop all the components whose value will be determined by other components and constraints, i.e. we can describe the different datapoints in a much smaller-dimensional space. However, the structure is in general *not* an Euclidean space (unless the structure is trivially an hyper-plane), but instead a *surface*, more in general a curved space or formally a *manifold*. Recall that we defined vector spaces in the Euclidean domain, so in general we cannot represent datapoints in the embedding space as vectors, and also keep the underlying structure: the sum of two vectors representing datapoints lying on the manifold will be a vector that does not lie on the manifold, just like naively summing two images of the Eiffel tower will not result in an image of the Eiffel tower.

Manifolds How can we deal with manifolds, but still keep the Euclidean formalism that was so useful to us?
Formally, an n-dimensional manifold, or n-manifold for short, is a *topological space* (a space with the additional concept of *neighborhood*) with the property that each point is *locally homeomorphic* to Euclidean space $\mathbb{R}^n$.
"Locally homeomorphic to Euclidean space" means that every point has a *neighborhood* homeomorphic to a neighborhood in Euclidean space, that is simply a n-ball, i.e. an hypersphere of a certain radius δ. Two neighborhoods are "homeomorphic" if there exists a *homeomorphism* between them, i.e. a smooth and invertible mapping from the points of the first to the points of the second. Notice that this means that points that are "close" on the manifold must be "close" also on the corresponding Euclidean space.
Informally, this means that at each point on the manifold, that is a curved space, we can locally think to lie on a *flat*, Euclidean space instead. The more we zoom out from the point, the more this approximation loses accuracy, but this is enough for us to define *locally* a Euclidean vector space. For instance, we live on the surface of Earth, a curved space, but locally we have the illusion of living on a flat space, so Euclidean distance makes sense to us, even though it is not globally correct (and in fact this must be accounted for when charting intercontinental courses, e.g. to go from A to B it may be shorter to go north from A to C and then south again to B instead of going straight from A to B).

In the following we will refer to curves and surfaces, as manifolds embedded in $\mathbb{R}^3$, since they are easier to visualize.

Differential geometry The study of local properties of curves and surfaces is the realm of *differential geometry*. The overall idea is that we can model mathematically a surface as a collection of neighborhoods (colored regions in fig. 8.5) and we require that the union of these regions should cover the entire surface and that each region can be mapped in a well-behaved way to a subset of $\mathbb{R}^2$ (Note that if we have a high dimensional surface the subset would be in some other dimension not necessarily two).

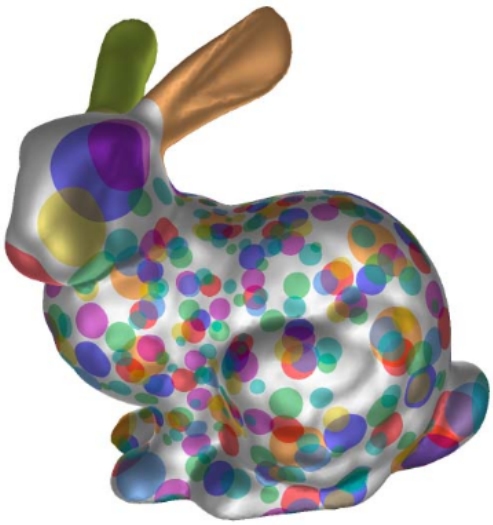


Figure 8.5: An example of differential geometry.

A differentiable manifold can be described using these maps, called coordinate *charts*. It is not generally possible to describe a manifold with just one chart, because the global structure of the manifold is different from the simple (Euclidean) structure of the charts. For example, no single flat map can represent the entire Earth without separation of adjacent features across the map's boundaries (although Russia and Alaska are adjacent, they appear separated on the map). Instead, we need several charts, collected in what is called an *atlas*.

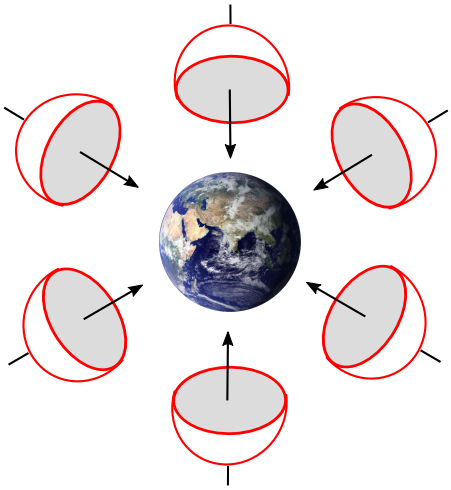


Figure 8.6: An atlas of e.g. 6 charts is able to correctly represent Earth. Notice two would suffice, but in general the minimum number is not trivial to compute.

Formally, a chart is a map

$$\phi : \mathbb{R}^2 \to \mathcal{S} \subset \mathbb{R}^3 \tag{8.24}$$

with the property of being

- invertible (we can go from a point on the surface to one in the plane and back);
- continuous (closeby points must be mapped by ϕ to closeby points).

If such mapping is also differentiable then we say that ϕ is a *diffeomorphism*; if it is infinitely-differentiable it is also said to be *smooth*.

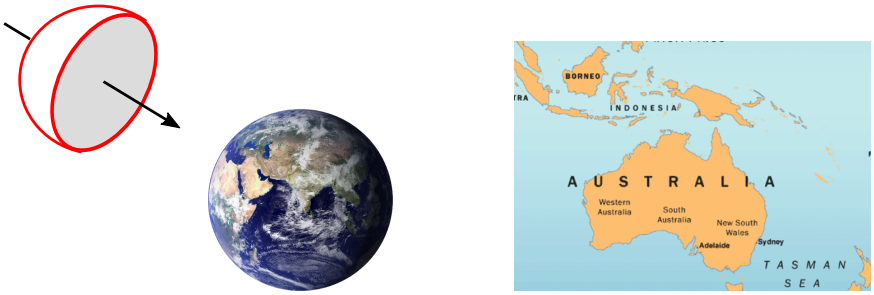


Figure 8.7: Chart representation. chart

General manifolds can be k-dimensional, meaning that we have charts:

$$\phi : \mathbb{R}^k \to \mathcal{M} \subset \mathbb{R}^d \quad \text{with } k < d. \tag{8.25}$$

Note that the choice of ϕ and the choice of the subset of the euclidean space is not unique.

Decoders as a chart The decoder of an autoencoder learns a mapping

$$D : \mathbb{R}^k \to \mathbb{R}^d \tag{8.26}$$

from a low-dimensional Euclidean space, the latent space $\mathbb{R}^k$ spanned by the codes $\mathbf{z}$, to a higher-dimensional embedding space, the data space $\mathbb{R}^d$ spanned by the observed datapoints $\mathbf{x}$. Such a mapping is differentiable, since it is defined by a neural network, and (in principle) is invertible via encoder E. Therefore, under the manifold hypothesis, the destination space is actually a manifold $\mathcal{M}$ and the mapping that the decoder is learning is a valid (parametric, via the network weights) chart for the manifold.

$$D : \mathbb{R}^k \to \mathcal{M} \subset \mathbb{R}^d \tag{8.27}$$

However, we do not know in advance the dimension of $\mathcal{M}$, i.e. the dimension of the Euclidean space $\mathbb{R}^k$ to which it is locally homeomorphic, therefore when designing autoencoders one should employ a trial and error strategy to decide the dimension k of the latent space, and see which choice yields the best results.

Limitations of autoencoders More often than not, we need to "hack" autoencoders a little bit to make them work well. In fact, ideally we would like, once an autoencoder is trained, for the decoder to have learnt *globally* the chart for the data manifold. However, what often happens in practice is that they overfit the data, leading to charts that perfectly map the codes from the latent space corresponding to training data to the the manifold, but when fed with unseen codes, they produce results that are clearly not on the manifold. For instance, a certain vector $\mathbf{z}_1$ might be the code for the image $\mathbf{x}_1$ of the digit 1, another vector $\mathbf{z}_2$ might be the code for another image $\mathbf{x}_2$ of the digit 2, but when taking as code the mean of the two $(\mathbf{z}_1 + \mathbf{z}_2)/2$ a decoder might produce garbage image, while from the *smoothness* requirement on the chart we would expect something that has to do with the digits 1 and 2.
There are many possible variations, acting as extra regularization to enforce *smoothness* in the chart (often referred to the latent space itself, i.e. a *smooth representation* of the latent space). For instance:

- *Denoising autoencoders*: set random values of the input to zero and require that the reconstructed data is exactly equal to the input, so we are adding some noise to the input. By doing this we obtain an autoencoder which can ignore that noise and it is forced to capture the correlation between the inputs.
- *Contractive autoencoders*: the idea is that we want the autoencoder to be robust to small variations and this is done by penalizing the gradient of latent code wrt the input.

Other autoencoders can be obtained adding constraint on the latent codes (*e.g.* sparsity), optimizing for the dimensions, etc.

8.3 Variational Autoencoders (VAE)

One fundamental limitation of autoencoders is their lack of guarantee about the regularity of the latent space. Indeed, we could be tempted to think that, if the latent space is regular enough (well "organized" by the encoder during the training process), we could take a point randomly from that latent space and decode it to get a new content, acting as a *generator*. However, it is pretty difficult (if not impossible) to ensure, a priori, that the encoder will organize the latent space in a smart way compatible with the generative process we just described. It could learn arbitrary functions, mapping similar inputs to arbitrarily distant regions of the latent space, without *coupling*.

This weakness is addressed by *Variational Autoencoders*, or *VAE* in short. A VAE does not learn a mapping from the data space to the latent space and its "inverse", but explicitly constructs the parameters of a *probability distribution* in the latent space, from which the latent codes are sampled.

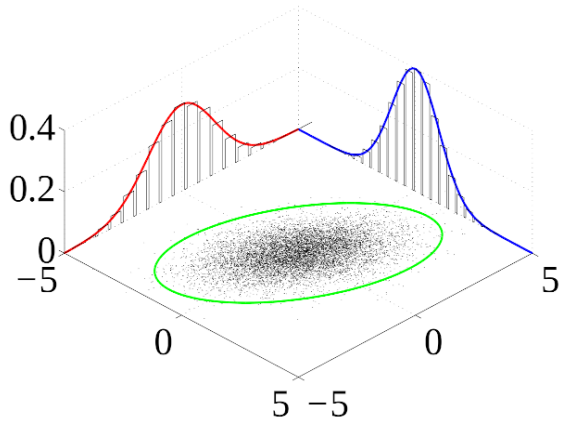


Figure 8.8: Variational Autoencoders with a multivariate Gaussian distribution.

The form of the distribution is fixed and decided *a priori*, for instance in the figure above we have chosen to represent the distribution of a 2-dimensional code $\mathbf{z}$ as a (multivariate) Gaussian.
For what follows, it is nice to have some recap on *information theory*: one is available at appendix B.
The encoder is not a deterministic parametric function anymore, but is a *probabilistic* parametric function, that models the parametric posterior distribution of the latent code $\mathbf{z}$, given the data $\mathbf{x}$

$$\overbrace{p_\theta(\mathbf{z}|\mathbf{x})}^{\text{probabilistic encoder}} = \frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{\underbrace{p_\theta(\mathbf{x})}_{\text{prior on the input data}}} = \frac{p_\theta(\mathbf{x},\mathbf{z})}{p_\theta(\mathbf{x})}. \tag{8.28}$$

This function is not deterministic since given the same input $\mathbf{x}$ twice, its output will not be the same $\mathbf{z}$, but rather it will be *sampled* from the parametric distribution that it encodes. In fact, one often says that the function "computes" the distribution.
To compute eq. (8.28), i.e. have a computable expression for the encoder, we need to be able to compute the prior on the input data $p(\mathbf{x})$, i.e. how the input data is distributed. However, the following expression

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})d\mathbf{z} \tag{8.29}$$

is a high-dimensional, *intractable* integral over all possible (continuous!) $\mathbf{z}$.
Since we cannot exactly compute this term, it seems like we cannot compute eq. (8.28) either. Indeed, the idea of variational autoencoders is to compute some approximation of that posterior distribution:

$$q_\phi(\mathbf{z}|\mathbf{x}) \approx p_\theta(\mathbf{z}|\mathbf{x}) \tag{8.30}$$

where $q_\phi(\mathbf{z}|\mathbf{x})$ is again a parametric probability distribution, with a *fixed* form, and will depend on the parameters ϕ of the neural network implementing the encoder.

Since we are looking for an approximation to p, we are going to ask that q and p are similar in the KL sense eq. (B.6), i.e. they have the same *information content*. So we compute an approximation

$$q_{\phi^*}(\mathbf{z}|\mathbf{x}) \approx p_\theta(\mathbf{z}|\mathbf{x}) \tag{8.31}$$

where ϕ^* is such that

$$\phi^* = \underset{\phi,\theta}{\operatorname{argmin}}\, KL(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})). \tag{8.32}$$

Once we obtain such a ϕ^*, we have the expression for our encoder. However, at the moment we cannot even write down a closed form expression for eq. (8.32) as $p_\theta(\mathbf{z}|\mathbf{x})$ is not given. However, with a somewhat long derivation we can simplify things:

$$\begin{aligned}
\phi^* &= \underset{\phi,\theta}{\operatorname{argmin}}\, KL(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) && (8.33)\\
&= \underset{\phi,\theta}{\operatorname{argmin}} - \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} && \text{(substituing the definition of KL)}\\
&= \underset{\phi,\theta}{\operatorname{argmin}} - \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})}\frac{1}{p_\theta(\mathbf{x})} && \text{(substituing the definition of p)}
\end{aligned}$$

Applying some properties of the logarithms we have:

$$\begin{aligned}
\phi^* &= \underset{\phi,\theta}{\operatorname{argmin}} - \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \left(\log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} + \log \frac{1}{p_\theta(\mathbf{x})} \right) && (8.34)\\
&= \underset{\phi,\theta}{\operatorname{argmin}} - \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \left(\log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} - \log p_\theta(\mathbf{x}) \right) && (8.35)
\end{aligned}$$

Splitting the summation we obtain:

$$\begin{aligned}
\phi^* &= \underset{\phi,\theta}{\operatorname{argmin}} - \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} + \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log p_\theta(\mathbf{x}) && (8.36)\\
&= \underset{\phi,\theta}{\operatorname{argmin}} - \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} + \log p_\theta(\mathbf{x}) \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) && (8.37)\\
&= \underset{\phi,\theta}{\operatorname{argmin}} - \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} + \log p_\theta(\mathbf{x}) && \text{(since the sum over all the z is equal 1)}
\end{aligned}$$

Because minimizing a function is equivalent to maximizing the negative of that function:

$$\phi^* = \underset{\phi,\theta}{\operatorname{argmax}} \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} - \log p_\theta(\mathbf{x}) \tag{8.38}$$

Now, until now we have not solved anything, since the term $\log p_\theta(\mathbf{x})$ is still an intractable term. However, it can be proven that the summation is always smaller than $\log p_\theta(\mathbf{x})$, i.e. it is a *lower bound* for that term. Basically, since we are maximizing the function, the most we can do is *tightening the gap* between the two terms, i.e. maximize the lower bound, so we relax the maximization problem and obtain:

$$\phi^* = \underset{\phi,\theta}{\operatorname{argmax}} \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \tag{8.39}$$

The sum is known as *Evidence variational Lower BOund*, so we can write:

$$\phi^* = \underset{\phi,\theta}{\operatorname{argmax}}\, ELBO_{\phi,\theta}(\mathbf{x}) \tag{8.40}$$

where

$$ELBO_{\phi,\theta}(\mathbf{x}) = \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \leq \log p_\theta(\mathbf{x}) \tag{8.41}$$

Let's now see how to maximize this term:

$$\max_{\phi,\theta} ELBO_{\phi,\theta}(\mathbf{x}) \tag{8.42}$$

$$= \max_{\phi,\theta} \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \tag{8.43}$$

$$= \max_{\phi,\theta} \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \tag{8.44}$$

Again, using some logarithms properties and dividing the summation we gain:

$$\max_{\phi,\theta} \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \left(\log p_\theta(\mathbf{x}|\mathbf{z}) + \log \frac{p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \right) \tag{8.45}$$

$$= \max_{\phi,\theta} \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log p_\theta(\mathbf{x}|\mathbf{z}) + \sum_{\mathbf{z}} q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \tag{8.46}$$

The first term is just the expected value of $\log p_\theta(\mathbf{x}|\mathbf{z})$ wrt $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$, while the second term represents a KL divergence, so we can write the previous equation as follows:

$$\max_{\phi,\theta} \underbrace{\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z})}_{\text{likelihood of } \mathbf{x} \text{ given } \mathbf{z}} - \underbrace{KL\left(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})\right)}_{\text{ensures } q_\phi(\mathbf{z}|\mathbf{x}) \approx p_\theta(\mathbf{z})}. \tag{8.47}$$

In this expression, also the other part of the architecture, the *decoder* shows up. In fact, the first term can be thought of as the goodness of the *decoder*, since it is the likelihood of a datapoint given a latent code $\mathbf{z}$ sampled by the distribution modeled by the *encoder* $q_\phi(\mathbf{z}|\mathbf{x})$.
On the other hand, the KL divergence term is what distinguishes VAEs from regular autoencoders. It ensures that the probabilistic encoder follows the distribution $p_\theta(\mathbf{z})$. Wait, but θ are the parameters of the posterior, so what is $p_\theta(\mathbf{z})$? In principle, it is the parametric prior over the latent codes $\mathbf{z}$, i.e. the distribution of the latent space. In practice, it is *not* parametric, since we have said in the beginning that we would have fixed the latent space distribution, and here is where we fix it.
Usually, one chooses to have the prior over the latent space to be a (multivariate) *Gaussian* with *no free parameters*, so we remove θ:

$$p_\theta(\mathbf{z}) = p(\mathbf{z}) = \mathcal{N}_{\mathbf{0},\mathbf{I}}(\mathbf{z}). \tag{8.48}$$

The idea is that the encoder will output some probability distribution over $\mathbf{z}$, that will be different for different datapoints $\mathbf{x}$, since it is a conditional probability. We will sample from this conditional probability to obtain an actual $\mathbf{z}$. Now, the KL term enforces that, *regardless of* $\mathbf{x}$, this $\mathbf{z}$ should be distributed in the latent space like eq. (8.48).

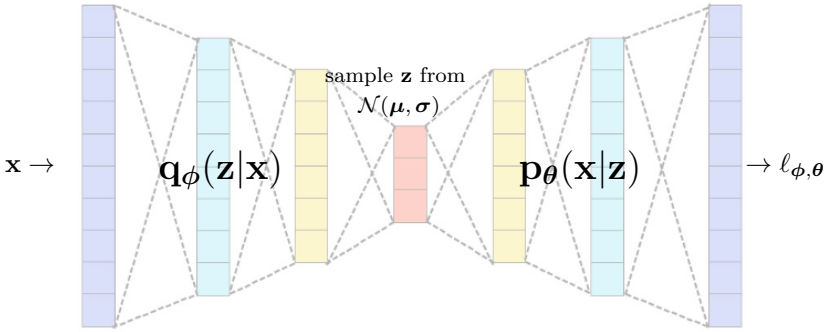


Figure 8.9: Variational autoencoder. The code $\mathbf{z}$ is sampled from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$, where $\boldsymbol{\mu}, \boldsymbol{\sigma}$ depend on the incoming datapoint $\mathbf{x}$ but globally follows $\mathcal{N}(\mathbf{0}, \mathbf{I})$.

Now, what expressions do the (probabilistic) encoder and decoder have? Up until now we have only said that they model probability distributions, but what exactly are they?
Somewhat confusing, the probabilistic encoder also generates a Gaussian distribution. **Beware**, it is **not** necessarily equal to the prior $p(\mathbf{z})$, since this is a *conditional* distribution, and hence depends also on the data $\mathbf{x}$:

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}_{\mu,\sigma}(\mathbf{z}). \tag{8.49}$$

In practice, the encoder produces the Gaussian mean μ and variance σ as a function of the input $\mathbf{x}$ and the network parameters ϕ, and then those are used to explicitly generate the distribution to sample $\mathbf{z}$ from. Notice, once again, that this distribution is conditional, since it depends on $\mathbf{x}$. Different regions of the latent space will correspond to different "types" of datapoints. Suppose we have two points $\mathbf{x}_1, \mathbf{x}_2$:

- for $\mathbf{x}_1$ the probabilistic encoder will generate a pair of parameters $\boldsymbol{\mu}_1, \boldsymbol{\sigma}_1$, defining a conditional Gaussian $\mathcal{N}_{\boldsymbol{\mu}_1,\boldsymbol{\sigma}_1}(\boldsymbol{z})$, and the various $\mathbf{z}_1$ we could sample from that Gaussian will be centered at the point $\boldsymbol{\mu}_1$ and vary in a certain small region around it, with variance $\boldsymbol{\sigma}_1$;
- for $\mathbf{x}_2$ the probabilistic encoder will generate a pair of parameters μ_2, σ_2, defining a conditional Gaussian $\mathcal{N}_{\boldsymbol{\mu}_2,\boldsymbol{\sigma}_2}(\boldsymbol{z})$, and the various $\mathbf{z}_2$ we could sample from that Gaussian will be centered at a *different* point $\boldsymbol{\mu}_2$ and vary in a certain *different* small region around it, with variance $\boldsymbol{\sigma}_2$.

However, both the codes $\mathbf{z}_1$ and the codes $\mathbf{z}_2$ should be distributed, *regardless* of what conditional Gaussian produced them and hence regardless of the input $\mathbf{x}$, like $p(\mathbf{z}) = \mathcal{N}_{\mathbf{0},\mathbf{I}}(\mathbf{z})$.

VAE: Training Now, to actually train a VAE, since we usually minimize loss functions instead of maximize objective functions, we put a minus sign in front of eq. (8.47) to get the *loss function of VAEs*:

$$\ell_{\phi,\theta} = \underbrace{-\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z})}_{\text{reconstruction error}} + \underbrace{KL\left(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})\right)}_{\text{regularizer}}. \tag{8.50}$$

The first term of the loss function represents the *reconstruction error*, just like in the regular autoencoder case; the second term represents the novelty of VAEs, a *regularizer* that enforces a prior on the distribution for the latent space thus enforcing smoothness and compactness in the learned space, so that the chart from the Euclidean space to the manifold of the data, learnt by the decoder, should be continuous even in regions of the latent space that are not encountered during training.

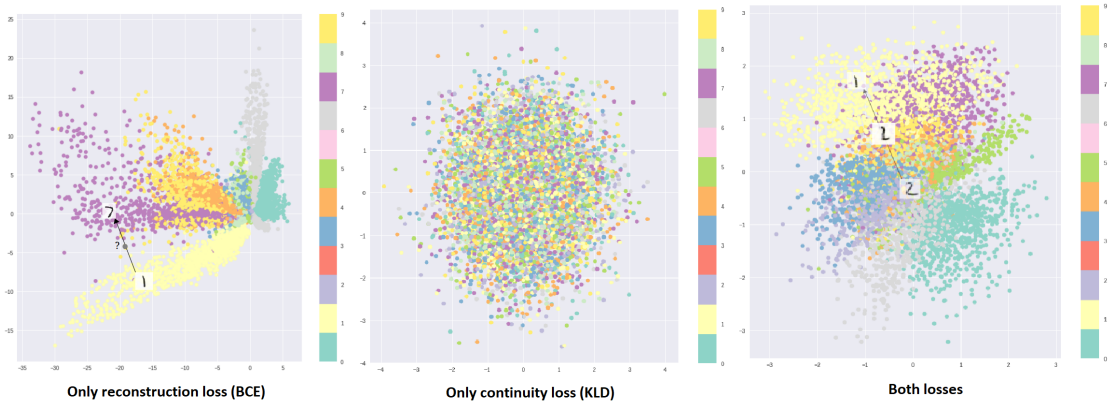


Figure 8.10: The role of the regularizer in VAEs, on the MNIST dataset. On the left we have no regularization, so regular autoencoders, in which the latent space is well behaved only in regions in which codes were assigned to observed data. In the center, only the regularization, so the codes are globally distributed as a normal Gaussian, but there is no correspondance with the input data, so we see codes for a digit mixed with codes for another digit. On the right, the combination of the two gives raise to a compact, smoothly changing distribution that has also localized regions (*modes*) corresponding to certain digits.

Note that the fact we are using two Gaussian distributions (in eq. (8.48) and in eq. (8.49)) allows us to have a closed form expression for the regularizer[1], and hence to write the loss in closed form.
When we train a VAE, the probabilistic encoder will "output" distributions $\mathcal{N}_{\mu,\sigma}$ from which we should sample to get a latent code to feed to the decoder. Recall that to train with a gradient descent-like algorithm we need to be able to backpropate gradients through the newtork, but this is a a problem, since in its naive form sampling from a random distribution is *not* a differentiable operation. Since we are sampling at training time we need the sampling procedure be differentiable because we need to backpropagate through the network.
To overcome this inconvenience, something called *reparametrization trick* is employed, that makes the sampling procedure differentiable. The idea is that in the computational graph there is now a node (intermediate variable) $\mathbf{z}$ which is a *random node*, something that backprop cannot flow through. However, we can make it so that $\mathbf{z}$ is not a *source of randomness*, but is a *deterministic function*

$$\mathbf{z} = g_\phi(\mathbf{x}, \boldsymbol{\epsilon}), \qquad \boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) \tag{8.51}$$

that relies on an *external source of randomness* $\boldsymbol{\epsilon}$.

[1] See https://arxiv.org/pdf/1907.08956.pdf for the expression and its derivation.

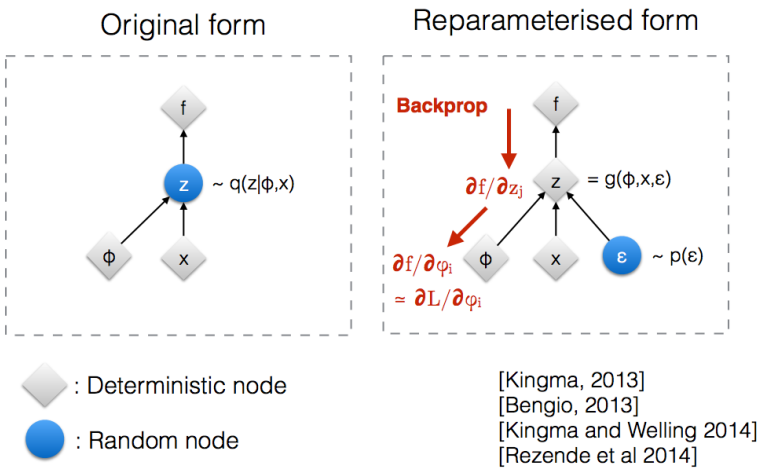


Figure 8.11: Reparametrization trick.

Usually this function looks like

$$\mathbf{z} = g_\phi(\mathbf{x}, \boldsymbol{\epsilon}) = \boldsymbol{\mu}_\phi(\mathbf{x}) + \underbrace{\boldsymbol{\sigma}_\phi(\mathbf{x}) \odot \boldsymbol{\epsilon}}_{\text{element-wise product}}, \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{8.52}$$

where we are taking an external source of randomness that follows a normal Gaussian, sampling a value from it, then deterministically shifting it and rescaling it with the mean and variances computed by the encoder, get back a

$$\mathbf{z} \sim \mathcal{N}_{\boldsymbol{\mu},\boldsymbol{\sigma}}(\mathbf{z}|\mathbf{x})$$

in a *deterministic* way. Where is the trick? We still need to sample from a distribution, something that is still not differentiable: there is still a random node in the computation graph. However, as you can see in fig. 8.11 now we do *not* need to backprop through this node, since the network parameters are on a different, separate path.

VAE: Testing When we are doing generation, we do not care about the encoding part of the VAE anymore and we only consider the decoder: we sample a latent code $\mathbf{z}$ according to eq. (8.48) and then we decode it to get a new, *generated* sample $\mathbf{x}$.
However, sampling on the boundaries of the learned distribution can still lead to the generation of unlikely samples, since the VAE probably has seen few samples that were mapped to a latent code in that region, that is therefore poorly modeled.

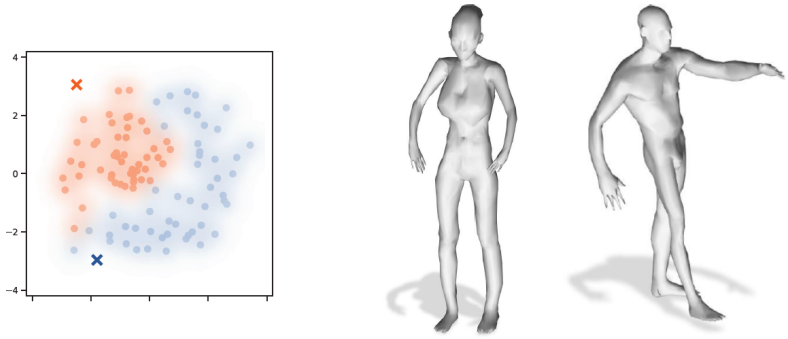


Figure 8.12: Sampling on the boundaries of the distribution.

Another relevant phenomenon is *posterior collapse*, in which the latents are ignored if the decoder becomes "too powerful", i.e. powerful enough to model the data perfectly, or if the encoder becomes "too weak". In fact, in this case $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ collapse to some constant values $\mathbf{a}, \mathbf{b}$, so that

$$q_\phi(\mathbf{z}|\mathbf{x}) \approx q_\phi(\mathbf{z}) = \mathcal{N}_{a,b}(\mathbf{z}), \tag{8.53}$$

the posterior of $\mathbf{z}$ *collapses* to some fixed Gaussian, regardless of the input. As a result, the decoder tries to reconstruct $\mathbf{x}$ by ignoring the uninformative $\mathbf{z}$ which are sampled from $\mathcal{N}_{a,b}(\mathbf{z})$.

VAE interpolation The regularity we induced in the latent space means that, once we have trained (correctly, which is hardly an easy task) a VAE, we can *interpolate* between samples. This means choosing two samples $\mathbf{x}_1, \mathbf{x}_2$, encode them to get their latent codes $\mathbf{z}_1, \mathbf{z}_2$, and then trace a line between $\mathbf{z}_1, \mathbf{z}_2$ and sample latent codes on the line. We are interpolating in the latent space, but if the decoder has correctly learnt a chart to the manifold of the data, by the continuity of the chart we should get samples that are close to the samples corresponding to the interpolating codes, effectively interpolating between samples.

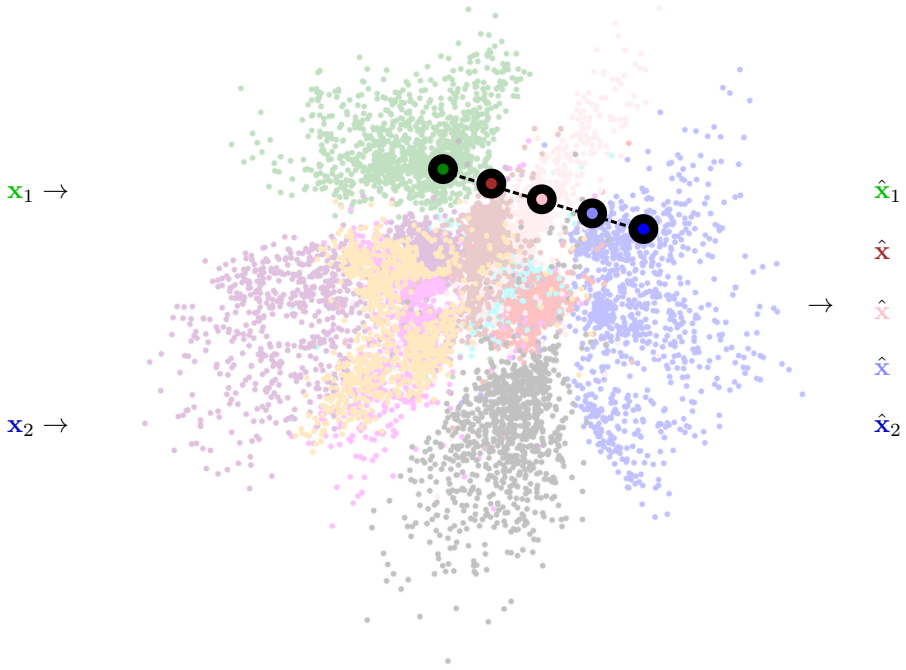


Figure 8.13: Variational Autoencoder interpolation.
If $\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2$ were pictures of two digits generated by the decoder, the interpolated samples $\hat{\mathbf{x}}$, $\hat{\mathbf{x}}$, $\hat{\mathbf{x}}$ should be pictures of "digits" that resemble both.

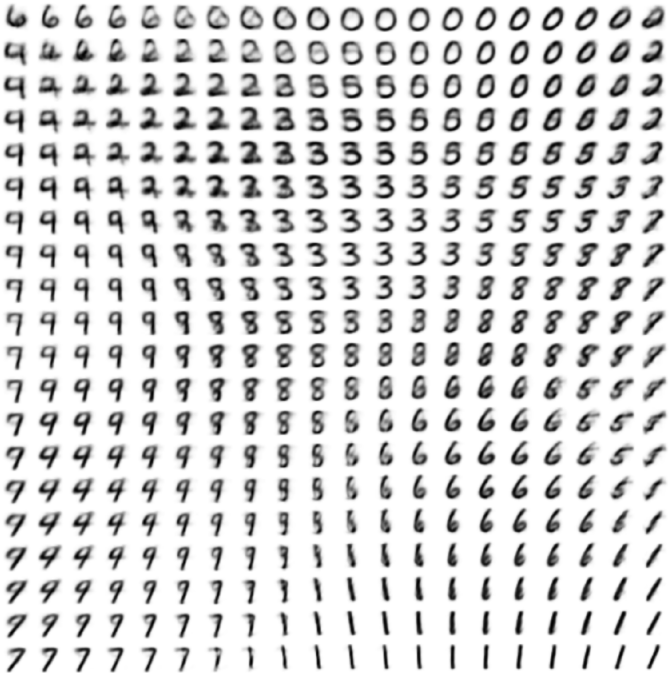


Figure 8.14: Interpolation between different images of digits.

Another interesting property of VAEs is that we can identify dimensions in the latent space that capture different semantics of the data, in particular we can do *disentanglement*: for example in fig. 8.15 one dimension represents the pose and the other dimension represents the identity of the person; then we can just interpolate along one dimension by setting a source point and a destination point and then change the dimensions spanning the expression.

Figure 8.15: Disentaglement.

Chapter 9

Geometric deep learning

Geometric deep learning (GDL) is an umbrella term for emerging techniques attempting to generalize (structured) deep neural models to *non-Euclidean* domains, such as graphs and meshes.

9.1 Introduction

Every task we have seen so far relied on the Euclidean domain. However, many scientific fields study data with an underlying structure that is a non-Euclidean space: think of social networks represented as graphs, or meshed surfaces in computer graphics.

Geometric deep learning vs manifold learning Notice that this setting is different from *manifold learning*, in which we seek for a manifold that justifies a given set of data. In fact, this data can still be represented as vectors in a (possibly high-dimensional) embedding space, and seeking a manifold is just a way for us to learn more accurate mappings (charts) from the latent space to the data space, subset of the embedding space.
On the other hand, this is not possible for data that *intrinsically* lives in a non-Euclidean domain such as a graph, because the information is encoded both as data on the domain (features of a certain user in a social network), but also in the *domain structure* (connections between users) itself, while in a Euclidean domain we care only about the information encoded by the data, since the domain has a shared, grid-like, flat structure for every possible type of data (*e.g.* images or audio signals).

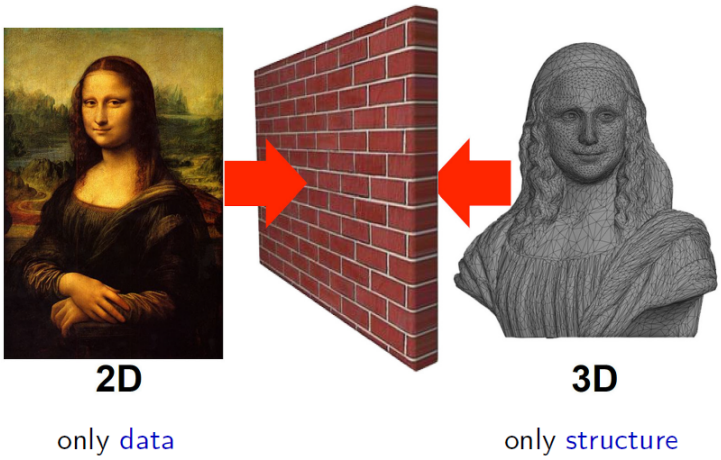


Figure 9.1: On the left, we have only data, i.e. the intensity values for the three channels for every pixel, that can be expressed as a vector-to-vector function. On the right, we have only the structure, but that still encodes relevant information, even without the data. We want to capture both.

So in this new settings we are not trying to *learn* a manifold, we already know the geometry of the manifold, or of the graph, the problem is that a great deal of the information of the data comes from this known, but hard to represent, geometric structure.
Another critical aspect to consider in geometric deep learning is the dynamic nature of the domains, so for example to make prediction on a social network we don't need to have all the information of how the graph changes over time (new social connections), instead for surfaces probably the information that we care about is encoded in how the object transforms.

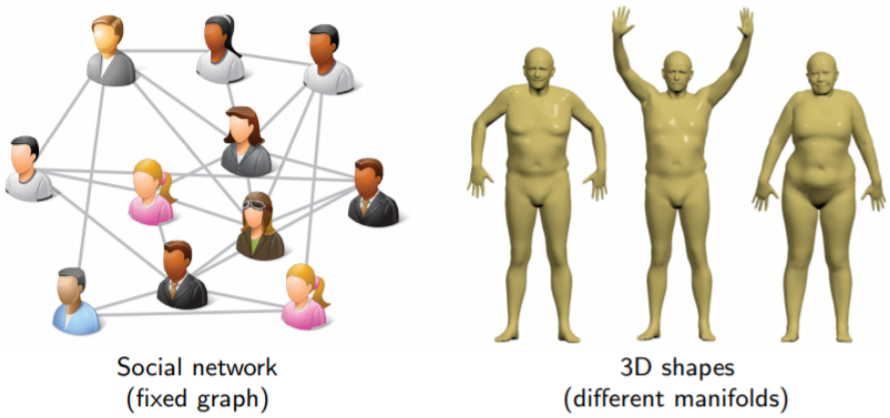


Figure 9.2: Static vs dynamic domain.

9.2 First examples

Computer vision has been one of the first applications of rudimentary GDL. Let's see some proposed solutions to problems dealing with meshes.
What is a mesh? In practice, when we deal with manifolds, we deal with discrete representations of them. One possible way to represent a surface is using a graph (we have a sampling of the vertices and then we connect neighboring vertices using edges). However, a surface has no "holes" between its vertices, but instead the edges are "filled", with a *face*. So, when we know that we are dealing with a surface that represents a real object, like a horse, we use a structure called *mesh* which is a collection of polygonal faces where we have vertices, edges and also we have faces. We want to have some constraints on the faces, and these constraints make up what is called manifold mesh.

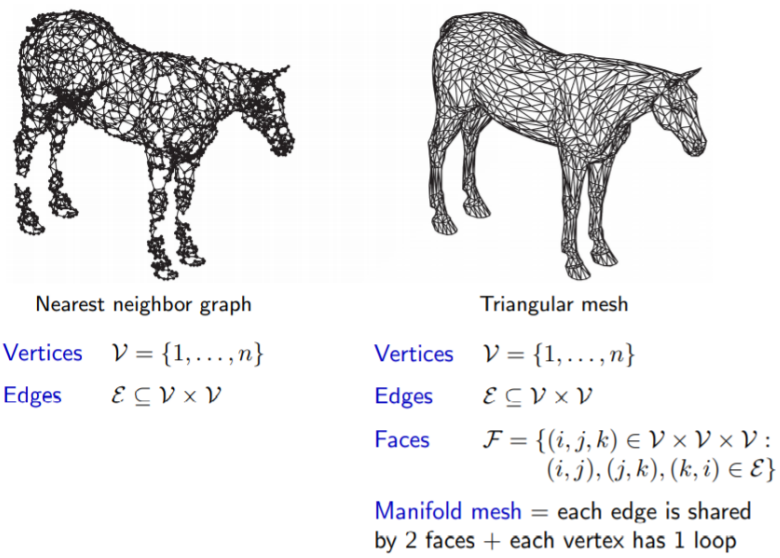


Figure 9.3: Discrete manifold.

Multi-view CNN Suppose we are given a dataset of $3D$ objects, and we want to classify them, e.g. to recognize furniture like tables, chairs and so on.
One approach that has been proposed to the task is called *multi-view CNN*. The idea is that we can treat the data as a collection of 2D images obtained just going around each object and taking a snapshot. Now, we can consider these images as items in our familiar Euclidean domain, and for instance employ the usual CNNs. However, since the images of an object all represent the same object, the networks should have some form of weight sharing, and at the end we would want to combine the prediction from each network in some way, for example using some kind of pooling.

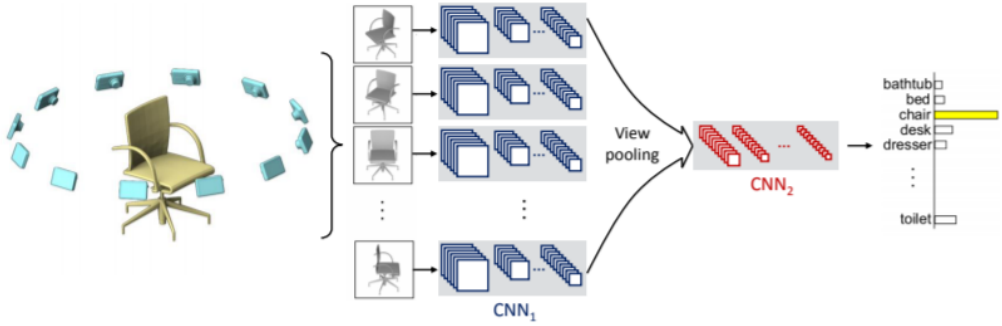
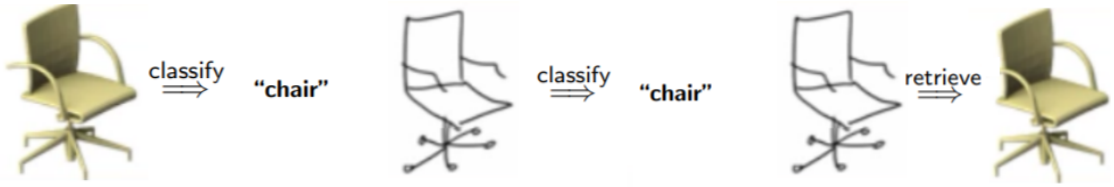


Figure 9.4: Multi-view CNN.

Other applications of this approach, in addition to the classification of 3D objects, are sketch classification (training the model with a different rendering) and sketch-based shape retrieval (rendering the different views of the objects).



(a) 3D shape classification and retrieval.

(b) Sketch classification.

(c) Sketch-based shape retrieval.

Figure 9.5: Application of multi-view CNN.

Note that this approach is not invariant to rotations in space, so it works well only if the object is rigid.

3D ShapeNets Another idea is representing a 3D object using *voxelization*, *i.e.* represent it as a collection of small 3D cubes, named *voxel*, the $3D$ counterpart of the $2D$ *pixel*. In this setting we can define a standard 3D convolution. Note that we are not looking at the surface but at the interior of the surface.

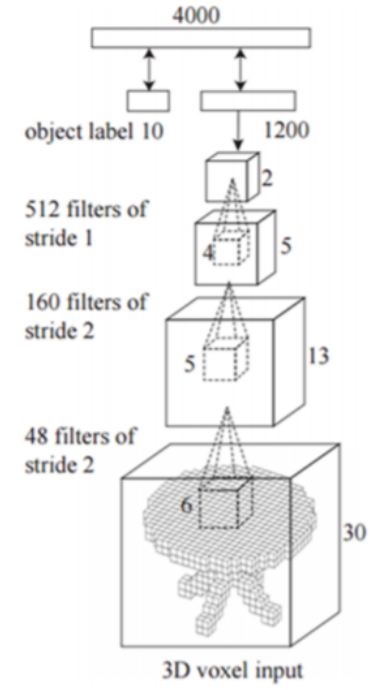


Figure 9.6: 3D ShapeNet.

Using 3D shape nets, just like in 2D neural network, we can learn hidden features, which we call 3D primitives, that are organized in a hierarchical fashion as we go deeper and deeper into the network, recognizing semantically valid features at the deepest level.

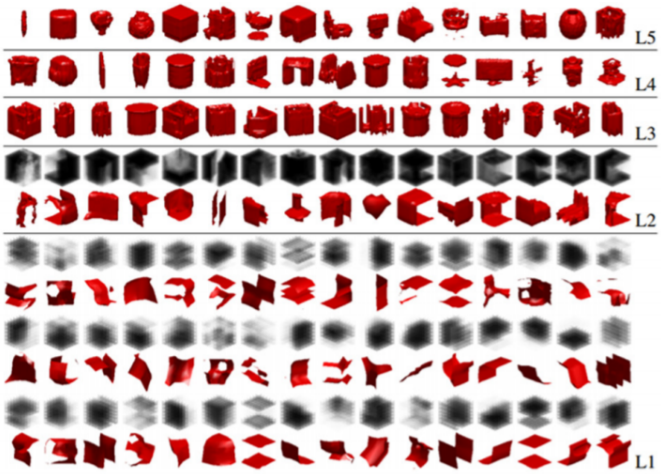


Figure 9.7: 3D primitives

9.3 Challenges

We have seen that we can do something when we approach the manifolds *extrinsically*, meaning operating in the $3D$ space where the object lives. However, such a filter would not be *deformation* invariant, since it will not bend along with the object.

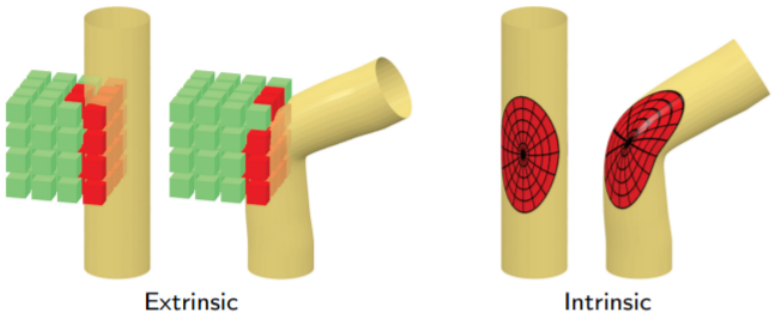


Figure 9.8: Extrinsic vs Intrinsic.

We would like for the filters to operate directly on the surface, so that they deform along with the object, and thus obtain deformation invariance *by construction*, since we have defined our convolutional kernel directly on the surface, not on the Euclidean domain.

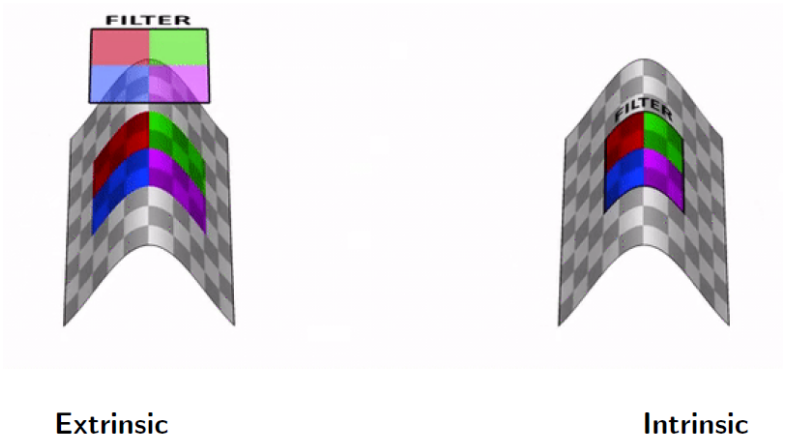


Figure 9.9: Extrinsic vs Intrisc.

But how to define such a non-Euclidean convolution?
Convolution relies on the grid-like structure of the Euclidean space, in which there is a well-behaved concept of distance and neighborhood. But how would you perform convolution on a graph? Suppose each node v has a scalar feature $x(v)$, just like each pixel i, j of an image has an intensity $f(i, j)$. In the case of an image we would convolve with a 3×3 kernel g like so

$$(g \star f)[i, j] = \sum_{m=1}^{3} \sum_{n=1}^{3} g[m, n] f[i - m, j - n] \tag{9.1}$$

in which we are exploiting the grid-like Euclidean structure to extract a *local patch* of *adjacent* pixels, then considered in a certain order (the indices of the summations) to properly combine them with the filter. But what would be a local patch for a node in a graph? Its immediate neighborhood? But then each node would have a neighborhood of different size, so we could not have weight sharing by using the same filter. Even if this was not already a problem, what would be such ordering for the neighborhood of a node?
Unlike images, there is no canonical ordering of the domain points in graphs and meshes, and the absence of this concept is a critical issue, named *local ambiguity*. In a graph there is no canonical way of visiting the neighbors of a node. In a mesh we have a notion of orientation, and from that we could label the neighbors of a node in a clockwise fashion, but then how would we fix the first node in the cycle, to begin our visit? This would still be ambiguous.

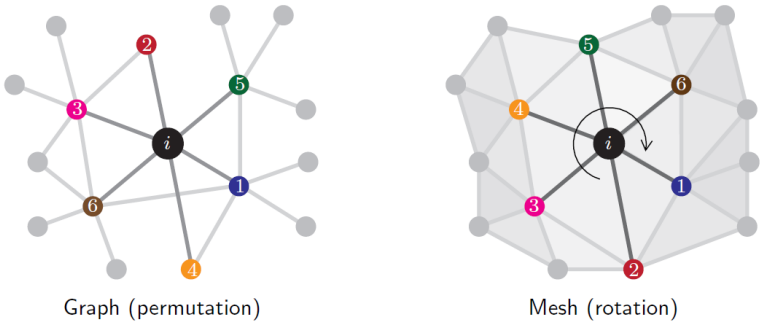


Figure 9.10: Local ambiguity.

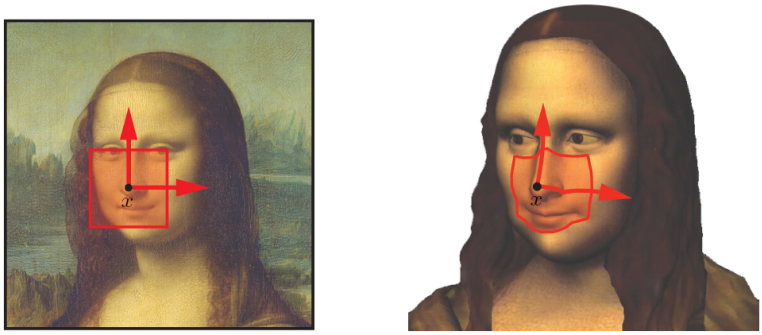


Figure 9.11: Idea of convolution on a mesh.

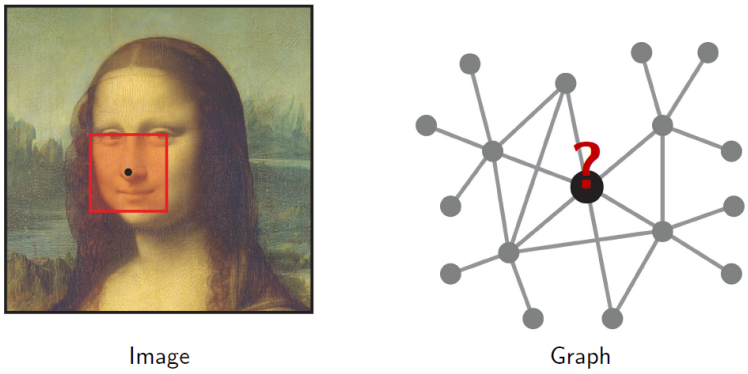


Figure 9.12: Not really clear how to define non-Euclidean convolution on graphs...

9.4 Generalize Convolution

In this section we will try to define convolution directly on manifolds, and we will see different approaches.

9.4.1 Global parametrization

A first approach to defining convolution on manifolds (in this case specifically mesh surfaces) is to map meshes to an intermidiate *parametric domain*, such as the $2D$ plane, where it is easier to operate.

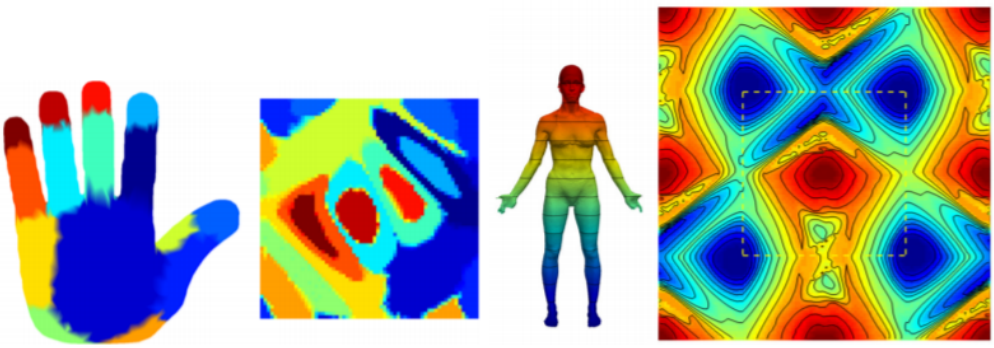


Figure 9.13: Global parametrization.

Everything that a neural network will learn in this domain will be valid in the original domain, since there exists an invertible mapping between the two domains. However you can imagine that such parametrizations are not unique, but there are many ways in which the same surface might be parametrized. Usually these maps introduce *distortion*, in terms of distances, relative angles, local areas not being preserved. Some of these maps will be "better" than others, in the sense that will guarantee *invariance* to some transformations, that if you recall is something we seek. Most notably, recall that a *defining property* of convolution is being *translation invariant*, so if we want to define a "proper" convolution, we must ensure that this property is satisfied.

How to define shift invariance on a surface? Recall that surfaces are manifolds, and manifolds are locally Euclidean, so locally to each point of the manifold we can define *vector fields* that encode a translation at all points on the surface, called *translation fields*. This means that at each point we will define a vector space, in which that point will be represented as some vector $\mathbf{x}$, and there will be a vector-to-vector function $\mathbf{y} = f(\mathbf{x})$ that will tell us, for each point in the local neighborhood of that point, what is the effect of the translation, i.e. the position vector $\mathbf{y}$ to which each point $\mathbf{x}$ is mapped after the translation.
However, there exists a theorem from differential geometry called the *hairy ball theorem* that states that for a general surface it is not possible to define translation fields that are smooth and also don't have singularities, i.e. points on the surface in which the field is ill-defined.

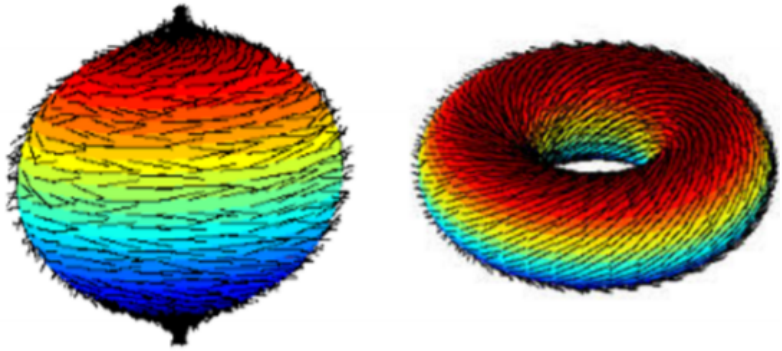


Figure 9.14: Hairy ball theorem. On a sphere we cannot have a translation field without singularities, meaning points in which the translation is not well defined. In this example they are the poles. For other surfaces, like the torus, instead we have no problems.

This means that we do not have any guarantee of being able to define convolution with this parametric approach, since in general we are not even able to define what translation is, so how can we define an operator that is translation invariant?
In fact, the *torus* is the only surface in which this is possible, since it is the only *closed orientable surface admitting a translational group*.

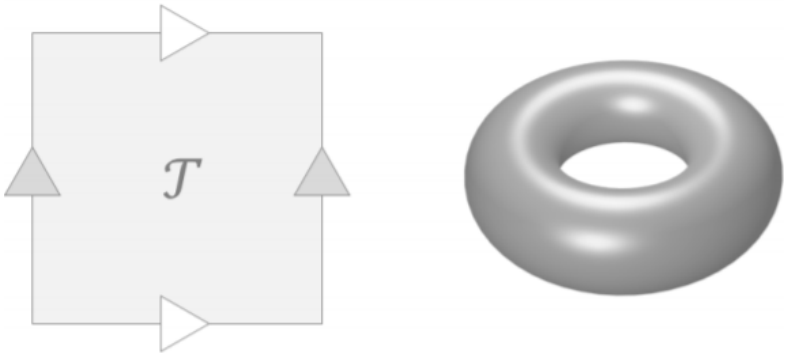


Figure 9.15: Parametrization of a torus.

9.4.2 Spectral convolution

We have tried to define convolution in terms of one of its defining properties (a property that the convolution and only the convolution satisfies), i.e. translation invariance. We have seen how this is not possible since the very notion of translation is ill-defined in manifolds.

Convolution in the Fourier domain Therefore, we now try to define convolution in terms of another one of its defining properties, in particular the *convolution theorem*. The convolution theorem states that the convolution is *diagonalized* by the *Fourier transform*, meaning that the Fourier transform of the convolution of two functions is simply the product of their Fourier transforms.

$$\widehat{(f \star g)} = \hat{f} \cdot \hat{g} \tag{9.2}$$

This is a first step, but simply transfers the problem, since a convolution involves a convolution integral (or sum) that in turn requires a Euclidean domain, but also the Fourier transform requires a similar integral (see appendix C for a more in depth discourse on the Fourier transform). However, there is another mathematical object that comes to our rescue: the *Laplacian* operator Δ.

The connection with the Laplacian The Laplacian operator is naturally defined in the Euclidean domain as a differential operator, but can be generalized also to other domains, including graphs and meshes.

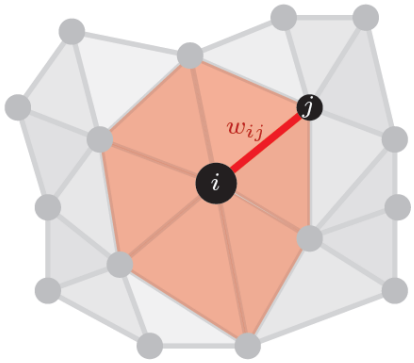


Figure 9.16: Neighborhood of a vertex.

The Laplacian acts locally on the neighborhood of each vertex. This is something that we like, since enforces locality by definition. Given a node signal $\mathbf{x}$, i.e. a function that associates features to each node i, the Laplacian associates to each node i the quantity

$$(\Delta \mathbf{x})_i = \sum_j w_{ij}(\mathbf{x}_i - \mathbf{x}_j). \tag{9.3}$$

The nice thing about the Laplacian, is that its *eigenfunctions* in the Euclidean domain are exactly the plane waves that make up the Fourier basis. What this means is that it is

$$\hat{f}(\xi) = \int_{\mathbb{R}} f(x) \underbrace{e^{-2\pi i x \xi}}_{\text{function of the Fourier basis}} dx \tag{9.4}$$

but also

$$\Delta \underbrace{\left(e^{-2\pi i x \xi}\right)}_{} = 4\pi^2 |\xi|^2 \underbrace{\left(e^{-2\pi i x \xi}\right)}_{}. \tag{9.5}$$

Since the Laplacian operator leaves the plane wave unchanged, it is a eigenfunction.
The fact that the Fourier basis of the Euclidean domain is composed by the Laplacian eigenfunctions allows us to generalize the Fourier transform to any non-Euclidean domain $\mathcal{X}$:

$$\hat{f}(\xi) = \sum_{k \geq 1} \underbrace{\int_{\mathcal{X}} f(x)\phi_k(x)dx}_{\langle f, \phi_k \rangle_{L^2(\mathcal{X})}} \tag{9.6}$$

where

$$\Delta\phi_k(x) = \overbrace{\lambda_k}^{\text{eigenvalue of the Laplacian}} \underbrace{\phi_k(x)}_{\text{eigenfunction of the Laplacian}}, \tag{9.7}$$

and

$$\langle \cdot, \cdot \rangle_{L^2(\mathcal{X})} \tag{9.8}$$

is the inner product defined on the functional (Hilbert) space of functions defined over the manifold $\mathcal{X}$. Just like the inner product of two vectors in vector spaces is the sum of the product of all their elements, the inner product of two functions in functional spaces is the "sum" (integral) of the product of all the values they take over the domain.

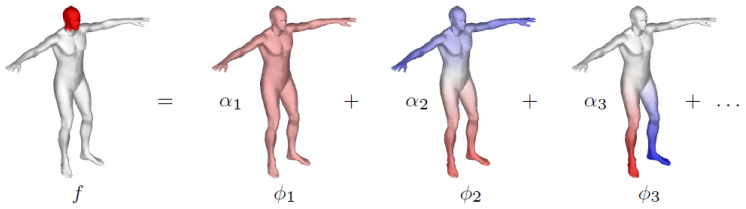


Figure 9.17: Decomposition of a function f over a manifold (the mesh of a humanoid) as the terms of a Fourier series. The contributions of every eigenfunction ϕ_i of the Laplacian over the Manifold (Fourier basis), weighted by the appropriate eigenvalue $\alpha_1 = \hat{f}_i$ (Fourier coefficient) make up the original function f.

Now, we are able to express our node signal $f : \mathcal{X} \to \mathbb{R}$ defined on a non-Euclidean domain $\mathcal{X}$ in the Fourier (or spectral) domain, and perform convolution with a (spectral) filter in the Fourier domain as a simple product.

$$(f \star g)(x) = \sum_{k \geq 1} \langle f, \phi_k \rangle_{L^2(\mathcal{X})} \underbrace{\langle g, \phi_k \rangle_{L^2(\mathcal{X})}}\, \phi_k(x) \quad \text{Fourier transform to obtain the spectral filter } \hat{g}_k \tag{9.9}$$

$$= \sum_{k \geq 1} \underbrace{\langle f, \phi_k \rangle_{L^2(\mathcal{X})}\, \hat{g}_k} \phi_k(x) \quad \text{Fourier transform to obtain } \hat{f}_k \tag{9.10}$$

$$= \sum_{k \geq 1} \underbrace{\hat{f}_k \hat{g}_k}\, \phi_k(x) \quad \text{convolution as a product in the Fourier domain} \tag{9.11}$$

$$= \sum_{k \geq 1} \underbrace{\widehat{(f \star g)}_k \phi_k(x)} \quad \text{inverse Fourier transform.} \tag{9.12}$$

Discrete spectral convolution Ok, but in practice we deal with discrete, not continuous objects. Our node signals does not vary continuously over the non-Euclidean domain $\mathcal{X}$, since this is actually a finite and discrete collection of nodes $\mathcal{V}$. Therefore the node signals are functions

$$f : \mathcal{V} \to \mathbb{R} \tag{9.13}$$

encoded as vectors

$$\mathbf{f} = \left(f_{v_1}, f_{v_2}, \ldots, f_{v_n}\right)^\top \tag{9.14}$$

that associate a scalar feature to each node $v \in \mathcal{V}$, with $|\mathcal{V}| = n$.

Note. If we had multiple features for each node, i.e. the node signal were multi-dimensional, i.e.

$$f : \mathcal{V} \to \mathbb{R}^m \tag{9.15}$$

then we would just encode this signal as a matrix

$$\mathbf{F} = \begin{pmatrix} \mathbf{f}_{v_1}^\top \\ \mathbf{f}_{v_2}^\top \\ \vdots \\ \mathbf{f}_{v_n}^\top \end{pmatrix} = \begin{pmatrix} f_{v_1}^{(1)} & \ldots & f_{v_1}^{(m)} \\ f_{v_2}^{(1)} & \ldots & f_{v_2}^{(m)} \\ \vdots & \ddots & \vdots \\ f_{v_n}^{(1)} & \ldots & f_{v_n}^{(m)} \end{pmatrix} \tag{9.16}$$

that associates a m-dimensional vector feature to each node $v \in \mathcal{V}$, with $|\mathcal{V}| = n$.

In this setting the Laplacian is an operator $\Delta(\cdot)$ that when applied to a node signal $\mathbf{x}$, represented as a n-dimensional vector, returns a n-dimensional vector whose i-th component is defined as in eq. (9.3). Hence, it is a $(n \times n)$ matrix $\boldsymbol{\Delta}$. Since it is a discrete object (a square matrix), we cannot talk about *eigenfunctions*, but instead we deal with the more familiar *eigenvectors*, that still share the fundamental property of being member of the Fourier basis in the non-Euclidean (now discrete) domain, for instance a graph. So we take an eigendecomposition of the Laplacian to yield

$$\boldsymbol{\Delta} = \boldsymbol{\Phi} \boldsymbol{\Lambda} \boldsymbol{\Phi}^\top, \tag{9.17}$$

with

$$\boldsymbol{\Phi} = \left(\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_n\right) \tag{9.18}$$

matrix of eigenvectors, and

$$\mathbf{\Lambda} = \text{diag}\{\lambda_1, \dots, \lambda_n\} \tag{9.19}$$

matrix of eigenvalues.
The Fourier transform eq. (9.6) now boils down to a simple matrix-vector product

$$\begin{aligned} \hat{f}(\xi) = \hat{\mathbf{f}} &= \sum_{k \geq 1} \langle \mathbf{f}, \boldsymbol{\phi}_k \rangle_{L^2(\mathcal{X})} \\ &= \sum_{k \geq 1} \mathbf{f}^\top \boldsymbol{\phi}_k \\ &= \sum_{k \geq 1} \boldsymbol{\phi}_k^\top \mathbf{f} \\ &= \mathbf{\Phi}^\top \mathbf{f}. \end{aligned} \tag{9.20}$$

As you can imagine, the inverse Fourier transform will be the inverse $(\mathbf{\Phi}^\top)^{-1}$, but since it is always possible to perform an eigendecomposition in which the eigenvectors form an orthonormal eigenbasis, then we can assume $\mathbf{\Phi}$ to be an orthogonal matrix, i.e. that it is

$$\mathbf{\Phi}^{-1} = \mathbf{\Phi}^\top \tag{9.21}$$

and therefore

$$(\mathbf{\Phi}^\top)^{-1} = (\mathbf{\Phi}^\top)^\top = \mathbf{\Phi}. \tag{9.22}$$

Therefore, the inverse Fourier transform will simply be

$$f(x) = \mathbf{f} = \mathbf{\Phi}\hat{\mathbf{f}}. \tag{9.23}$$

In this discrete setting, performing a spectral convolution of a node signal $f(x)$ encoded as a vector $\mathbf{f} = (f_1, \dots, f_n)^\top$ with a filter $g(x)$ encoded as a vector $\mathbf{g} = (g_1, \dots, g_n)^\top$ means

$$\begin{aligned} \mathbf{f} \star \mathbf{g} &= \mathbf{\Phi} \left[\left(\mathbf{\Phi}^\top \mathbf{g} \right) \odot \left(\mathbf{\Phi}^\top \mathbf{f} \right) \right] \\ &= \mathbf{\Phi} \underbrace{\left(\hat{\mathbf{g}} \odot \hat{\mathbf{f}} \right)}_{\text{convolution as a (element-wise) product in the Fourier domain}} \\ &= \mathbf{\Phi} \begin{pmatrix} \hat{g}_1 \cdot \hat{f}_1 \\ \vdots \\ \hat{g}_n \cdot \hat{f}_n \end{pmatrix} \\ &= \mathbf{\Phi} \underbrace{\begin{pmatrix} \hat{g}_1 & & \\ & \ddots & \\ & & \hat{g}_n \end{pmatrix}}_{\text{spectral filter}} \begin{pmatrix} \hat{f}_1 \\ \vdots \\ \hat{f}_n \end{pmatrix} \\ &= \underbrace{\mathbf{\Phi} \widehat{(f \star g)}}_{\text{inverse Fourier transform}}. \end{aligned} \tag{9.24}$$

Limitations of the spectral convolution Is this newly defined convolution a proper convolution?

The spectral convolution is not shift invariant One of the defining properties of convolution is shift invariance. Have we been able to recover this property, with this new approach? Recall that in the Euclidean (discrete) domain the shift invariance property of convolution comes from the peculiar structure of the matrix encoding the operation:

$$\mathbf{f} \star \mathbf{g} = \underbrace{\begin{pmatrix} g_1 & g_2 & \cdots & \cdots & g_n \\ g_n & g_1 & g_2 & \cdots & g_{n-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ g_2 & g_3 & \cdots & \cdots & g_1 \end{pmatrix}}_{\text{circulant matrix}} \mathbf{f}. \tag{9.25}$$

The fact that convolution in the Fourier domain is a simple product means that the Fourier basis (that as we have seen is composed by the Laplacian eigenvectors) *diagonalizes* the circulant matrix:

$$\begin{pmatrix} g_1 & g_2 & \cdots & \cdots & g_n \\ g_n & g_1 & g_2 & \cdots & g_{n-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ g_2 & g_3 & \cdots & \cdots & g_1 \end{pmatrix} = \boldsymbol{\Phi} \begin{pmatrix} \hat{g}_1 & & \\ & \ddots & \\ & & \hat{g}_n \end{pmatrix} \boldsymbol{\Phi}^\top = \boldsymbol{\Phi}\, \mathrm{diag}\{\hat{g}_1, \dots, \hat{g}_n\}\, \boldsymbol{\Phi}^\top, \tag{9.26}$$

and this is another way of seeing the derivation done in eq. (9.24). However, for the spectral convolution we have defined to be shift invariant, we should be able to express it as a matrix-vector product in which the matrix has a circulant structure, as in eq. (9.25). But it turns out that in

$$\begin{aligned} \mathbf{f} \star \mathbf{g} &= \underbrace{\boldsymbol{\Phi}\, \mathrm{diag}\{\hat{g}_1, \dots, \hat{g}_n\}\, \boldsymbol{\Phi}^\top}_{\mathbf{G}} \mathbf{f} \\ &= \mathbf{G}\mathbf{f} \end{aligned} \tag{9.27}$$

the matrix $\mathbf{G}$ does *not* have a circulant structure, so the spectral convolution is *not* shift-invariant.

Spectral convolution is domain dependent Convolution is domain dependent. In fact, recall that the Fourier basis that allows us to generalize the Fourier transform to any domain, and so to perform the convolution, is composed by the eigenfunctions $\phi_1, \dots, \phi_n$ of the Laplacian, that *depend on the domain*.
This is true also with regular Euclidean convolution, in which the Fourier basis are the plane waves $e^{-2\pi i \mathbf{x} \cdot \boldsymbol{\xi}}$. We never thought about this as a problem, since all our data, like images, live in the Euclidean domain $\mathbb{R}^n$, properly discretized to obtain images represented as pixels.
However, in this new setting being domain dependent means that if we employed spectral convolution (as defined so far) in a model, composed by layers that would learn the proper spectral filters $\hat{\mathbf{g}}$ just like regular Euclidean convolutional layers would do, then this model would be intrinsecally bounded to that specific domain. Imagine the domain to be a graph. We have developed a very powerful model to perform some inference on the graph. Now, after training, we would like to test our model on a new, unseen graph. Well, the model is useless, since that is another, completely different domain, in which the Laplacian acts differently and hence has different eigenfunctions leading to different Fourier basis. The spectral convolution in this new domain is *defined differently*, so the spectral filters we spent so much time learning at training time on a different graph are of no use here.

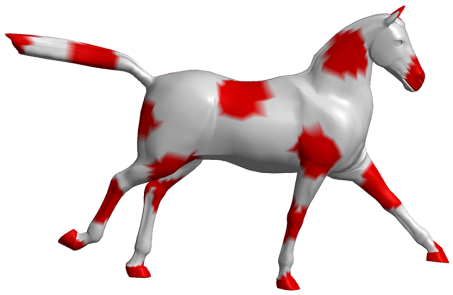


Figure 9.18: Signal $\mathbf{x}$ over the horse's mesh.

Imagine this scenario, now on a mesh. We have trained our (spectral) CNN to "detect edges" of some node signal $\mathbf{x}$ (as you can see in fig. 9.18) on a mesh, for instance of a horse, and learned a spectral filter $\hat{\mathbf{Y}}$ so that on this mesh, that is one specific domain with its specific Fourier basis $\mathbf{\Phi}$, the spectral convolution

$$\mathbf{\Phi}\hat{\mathbf{Y}}\mathbf{\Phi}^\top\mathbf{x} \tag{9.28}$$

seems to be producing nice results, as shown in fig. 9.19.

Figure 9.19: Edge detection on a mesh.

So, we would imagine that our model was still able to detect edges if the mesh was animated, e.g. with the horse running, since our filter is now *intrinsic*, it operates directly on the manifold and not on the embedding space and so it deforms with the mesh and therefore it should not be affected. However, at each frame the mesh deforms into a new mesh, that is a new domain with a new Fourier basis $\mathbf{\Psi}$. Therefore, our edge detection model now computes

$$\mathbf{\Psi}\hat{\mathbf{Y}}\mathbf{\Psi}^\top\mathbf{x} \tag{9.29}$$

since it has no choice, the very *operation* of convolution is defined differently in this new domain, the model can only choose what spectral filters to use.

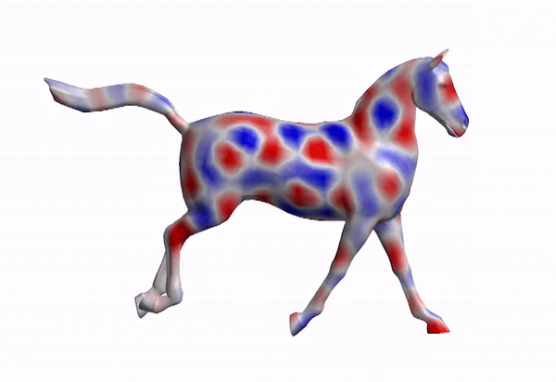


Figure 9.20: As you can imagine, our nice edge detection model fails miserably

Spectral convolution is not local As we have defined it, we have no guarantee that spectral convolution will be local in space. We can visualize spatially the learnt spectral filters

$$\text{diag}\{\hat{g}_1, \dots, \hat{g}_n\} \tag{9.30}$$

by going back to the spatial domain with an inverse Fourier transform

$$\mathbf{\Phi}\text{diag}\{\hat{g}_1, \dots, \hat{g}_n\}. \tag{9.31}$$

When going back to the spatial domain, we see a spatial representation of the convolutional filters. Usually, these filters are not local at all, which instead is what we would like to have.

Variants of the spectral convolution Of course these are *huge* limitations, and several works in recent years have tried to overcome them.

Spectral CNN with smooth spectral filters This technique modifies the definition of spectral convolution to try and overcome the problem of non-locality. Instead of learning a spectral filter $\text{diag}\{\hat{g}_1, \dots, \hat{g}_n\}$ to perform the spectral convolution

$$f \star g = \mathbf{\Phi}\ \text{diag}\{\hat{g}_1, \dots, \hat{g}_n\}\ \mathbf{\Phi}^\top \mathbf{f} \tag{9.32}$$

as we have defined it so far, we now learn some *smooth transformation* $\tau(\mathbf{\Lambda})$ acting on the eigenvalues $\mathbf{\Lambda}$ of the Laplacian. So the values $\hat{g}_1, \dots, \hat{g}_n$ cannot be arbitrary values, but must be the eigenvalues $\lambda_1, \dots, \lambda_n$ of the $\mathbf{\Delta}$, transformed in some way. We want to learn this transformation. Of course, if we allowed any transformation to be learned, then we would go back to having arbitrary values $\hat{g}_1, \dots, \hat{g}_n$. Instead, it turns out that by restricting the transformation $\tau(\cdot)$ to be *smooth*, we obtain the *localization in space* that we wanted.
This result is the *Parseval's identity*, an important result in signal processing theory. In the standard Euclidean setting it states

$$\int_{-\infty}^{+\infty} |x|^{2k} |f(x)|^2 dx = \int_{-\infty}^{+\infty} \left| \frac{\partial^k \hat{f}(\omega)}{\partial \omega^k} \right|^2. \tag{9.33}$$

On the left hand side we have the spatial domain, in the spatial variable x. If the integral converges for a certain k, then this provides a measure of the *locality* of the function in space.

On the right hand side we have the frequency (or spectral, or Fourier, they are all synonyms) domain, defined in terms of the *angular frequency* variable $\omega = 2\pi\xi$. If the integral converges for a certain k, then this provides a measure of the *smoothness* of the function in frequency.
The two concepts are the same. So, extending this identity to the non-Euclidean setting, this means that if we want our filters $\tau(\boldsymbol{\Lambda})$ to be localized in space, we must require $\tau(\cdot)$, that operates in frequency, to be smooth.
We choose the transformation to be a *linear combination of smooth functions*, in which the coefficients of the linear combination will be the *learnable parameters*.

$$\tau_{\boldsymbol{\alpha}}(\lambda_k) = \sum_{j=1}^{r} \alpha_j \beta_j(\lambda_k) \tag{9.34}$$

in which $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_r)^\top$ is the vector of filter (learnable) parameters, and $\beta_1(\lambda), \dots, \beta_r(\lambda)$ are *smooth kernel functions*. This works well since a linear combination of smooth functions is also a smooth function. Now, we can choose the various kernel functions to be any smooth functions (e.g. polynomials), and this choice becomes a hyperparameter of the model.

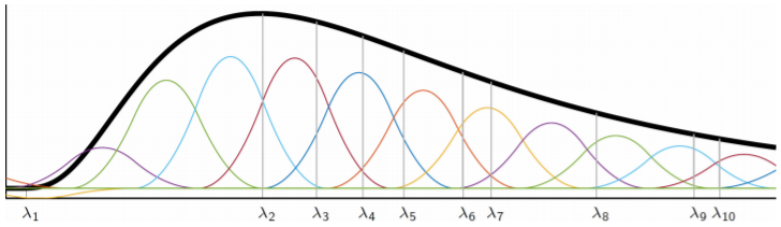


Figure 9.21: Linear combination of 10 Gaussian kernel functions, that yields a smooth transformation of the Laplacian eigenvalues. This will translate to a localized filter.

So we are parametrizing our spectral filter to be a smooth *parametric* filter $\tau_{\boldsymbol{\alpha}}(\cdot)$, so that the convolutional filter will operate to compute

$$\begin{aligned} \mathbf{f} \star \mathbf{g} &= \boldsymbol{\Phi} \underbrace{\operatorname{diag}\{\hat{g}_1, \dots, \hat{g}_n\}}_{\tau_{\boldsymbol{\alpha}}(\boldsymbol{\Lambda})} \boldsymbol{\Phi}^\top \mathbf{f} \\ &= \boldsymbol{\Phi} \begin{pmatrix} \tau_{\boldsymbol{\alpha}}(\lambda_1) & & \\ & \ddots & \\ & & \tau_{\boldsymbol{\alpha}}(\lambda_n) \end{pmatrix} \boldsymbol{\Phi}^\top \mathbf{f} \end{aligned} \tag{9.35}$$

If we write

$$\tau_{\boldsymbol{\alpha}}(\lambda_k) = \sum_{j=1}^{r} \alpha_j \beta_j(\lambda_k) = (\mathbf{B}\boldsymbol{\alpha})_k \tag{9.36}$$

where $\mathbf{B}$ is a $(n \times r)$ matrix result by a column-by-column stacking of the r smooth kernels β defined on every one of n eigenvalues, then we can write eq. (9.35) as

$$\boldsymbol{\Phi} \begin{pmatrix} (\mathbf{B}\boldsymbol{\alpha})_1 & & \\ & \ddots & \\ & & (\mathbf{B}\boldsymbol{\alpha})_n \end{pmatrix} \boldsymbol{\Phi}^\top \mathbf{f} = \boldsymbol{\Phi}\mathbf{W}\boldsymbol{\Phi}^\top \mathbf{f} \tag{9.37}$$

where $\mathbf{W} = \text{diag}(\mathbf{B}\boldsymbol{\alpha})$ is the weight matrix of the layer, in which $\mathbf{B}$ is a fixed, domain-dependent transformation and $\boldsymbol{\alpha}$ is the parameters to be learned. Recall that these are r scalar values, so the number of parameters *does not depend on the input size*, therefore we have $O(1)$ parameters per layer, similarly to the Euclidean convolution. In fact, we can think of r as kernel size of a Euclidean convolutional filter.

Note. This technique manages to rescue locality, but the problem of domain independence still remains.

9.4.3 Spatial convolution

As we have seen, defining convolution in the Fourier domain has an inherent drawback of inability to adapt the models across different domains. We will therefore need to resort to an alternative generalization of the convolution, now directly in the *spatial* domain, i.e. the domain in which data naturally lives. However, even this second operation will not be a "proper" convolution.
Many recent works have proposed their approaches to defining convolution on general surfaces. However, basically all of them have to give up on translation invariance, and define some convolution-like operators that shares some other properties with convolution.
On a Euclidean domain, due to shift-invariance the convolution can be thought of as passing a template at each point of the domain and recording the correlation of the template with the function (e.g., an image) at that point (e.g., a pixel). Thinking of image filtering, this amounts to extracting a patch of pixels, multiplying it element-wise with a template (the filter) and summing up the results, then moving on to the next position in a sliding window manner. Shift-invariance implies that the very operation of extracting the patch at each position is always the same.
However, we have seen how on non-Euclidean domains we cannot even define correctly the notion of shift, let alone shift-invariance of operators. Therefore, extracting a local "patch" would be position-dependent. Furthermore, we have seen how it is difficult to globally parametrize a manifold, therefore it is not clear how to define *adjacency* to even extract a *local* patch. Given a certain point, what is local and what is not?
So, one of the properties that we want to recover is *locality*. So, for each node i, we could define a local system of (polar) coordinates $\mathbf{u}_{ij}$ in which node i is the origin, and every other point (node or point in the face) is identified in terms of radius and angle wrt an axis going from i to some other point j.

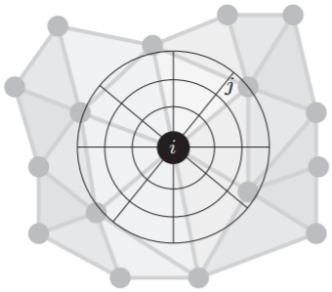


Figure 9.22: Local system of coordinates.

Now, how many nodes should we consider when extracting a *local* patch? To enforce locality we should give more weight to closer nodes and less to further nodes. This is done with exponentially decaying local weights defined by means of several Gaussians, according to some quantization.

$$w_{i,j} = exp\left(-\left(\mathbf{u}_{ij} - \mu\right)^\top \mathbf{\Sigma}^{-1} \left(\mathbf{u}_{ij} - \mu\right)\right) \tag{9.38}$$

The mean and variances of these Gaussians are learnable, meaning that the model can learn that some nodes have more influence than others in the neighborhood of each node.

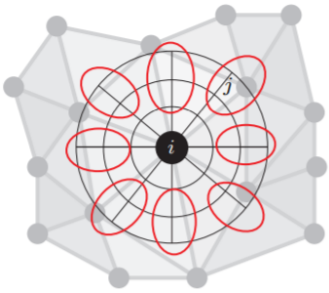


Figure 9.23: Local system of coordinates.

So in this system of coordinates we can now express a *localized node signal*, that is a function

$$f : \mathcal{V} \to \mathbb{R}^d \tag{9.39}$$

that associates each node to its d features, and extracting a patch amounts to averaging the function f by these weights, at each point (in the local system of coordinates) where the nodes are localized.
In the same way we can represent also a localized *filter*, i.e. express a function

$$g : \mathcal{V} \to \mathbb{R} \tag{9.40}$$

that will associate to each point an importance, again multiplied by the proper Gaussian weights. Notice that regular convolutional filters enforce locality by being smaller than the image, i.e. giving zero importance to pixels that are outside their range. Here we cannot do the same, so we define the filter on every node, then transform it locally to cover every point in order to enforce locality by means of the same Gaussian weights we used for the node signal.
So then, convolution is defined as the sum of the element-wise products between these weighted, localized node signal and filter:

$$(f \star g)_i = \mathbf{f}_i^\top \mathbf{g}, \qquad i = 1, \ldots, d \tag{9.41}$$

just like we would convolve a regular convolutional filter and a local patch.
Notice that there is no reason to enforce Gaussian weights specifically, instead of some other weighting mechanism.

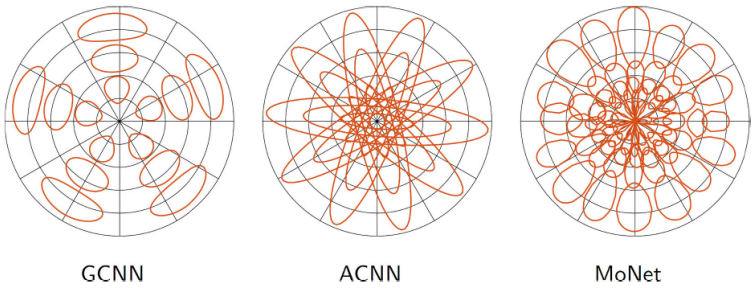


Figure 9.24: Different possible weighting mechanism to localize functions in the neighborhood of a point.

Many other approaches of this kind are possible, in which one defines convolution spatially as some form of *aggregation* of the node signals from the neighborhood of each node, then *combined* in some way with the representation of the node itself.
Suppose we have an intermediate hidden representation $\mathbf{h}_v^{(k-1)}$ for each node v in the graph, at input to the k-th convolutional layer. Then the output $\mathbf{h}_v^{(k)}$ will be

$$\begin{aligned} \mathbf{a}_v^{(k-1)} &= \text{AGGREGATE}_k\left(\{\mathbf{h}_u^{(k-1)},\ u \in \mathcal{N}(v)\}\right) \\ \mathbf{h}_v^{(k)} &= \text{COMBINE}_k\left(\mathbf{a}_v^{(k-1)}, \mathbf{h}_v^{(k-1)}\right). \end{aligned} \tag{9.42}$$

where the $\text{AGGREGATE}_k(\cdot)$ function has the role to enforce locality for this to be a well-behaved convolution-like operator.

Chapter 10

Adversarial training

Adversarial machine learning is an umbrella term that refers to a class of methods that, with different motivations, seek to fool models by supplying deceptive input.
This can be done to test the robustness of a model, to probe the level of understanding the network has of the underlying task, by looking at the error rate on examples that are intentionally constructed to be difficult to process by the model, called *adversarial examples*. These examples are generated by using an optimization procedure to search for an input x' near a data point x such that the model output is very different at x'. In many cases, x' can be so similar to x that a human observer cannot tell the difference between the original example and the adversarial example, but the network can make highly different predictions.
On the other hand, one can perform *adversarial training*, that involves two models being trained *in competition* with each other, i.e. as *adversaries*. This is the topic on which we will focus.

10.1 Generative Adversarial Networks

Generative adversarial network (GAN) is a class of machine learning frameworks that aim at generating realistic data by adversarial training.

10.1.1 Introduction

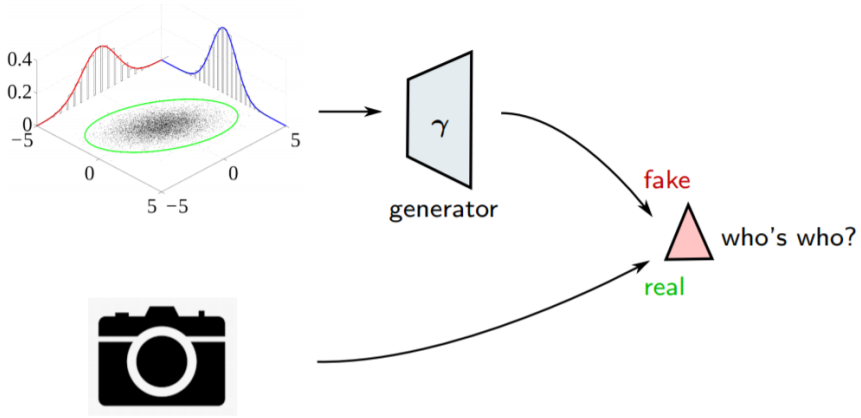


Figure 10.1: Decoder as a generator of fake samples.

Let's consider the situation depicted in fig. 10.1. We have the decoder part of a VAE, parametrized by some parameters γ. This decoder acts as a *generator*, since we can sample a random *latent code* from the probability distribution defined over the latent space, supply it to the decoder that will decode it into a new (generated) sample in data space. How good is this generator at producing new samples? Our criterion is the *realism* of the sample, i.e. how similar and indistinguishable it is from samples taken from real data, like real images. We will say that the sample produced by the generator is *fake*, since it does not come from the real data distribution, while the real samples are, well, *real*. Ideally, we would like our generator to be so good that no one could tell fake samples (generated by it) from the real samples apart.
Now, we would like to have a score that can be interpreted as the probability that a sample is real or not, to measure the goodness of the generator. However, how can we define such *score*? Visual inspection is not enough of course. The key idea of *Generative Adversarial Networks* (GAN) is that we can actually train a model, called *discriminator*, to distinguish between real and fake samples.

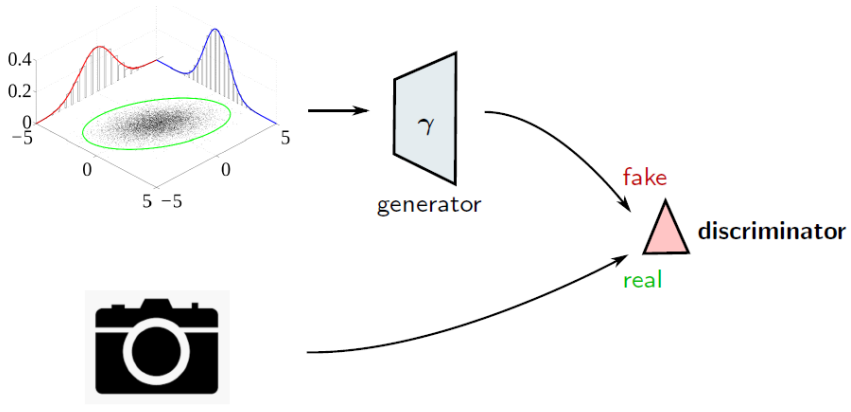


Figure 10.2: GAN.

We want to train a generator in such a way that a discriminator cannot distinguish between its generated sample and the real one. Simultaneously, we want to train a discriminator which is able to distinguish between fake samples and real ones. So, the generator and the discriminator are competing in a game: the generator tries to fool the discriminator and the discriminator tries to improve its proficiency in distinguishing forgeries from real data. They are *adversaries*.
Both are implemented as deep neural networks, the generator parametrized by some parameters γ, and the discriminator by some parameters δ. The task on which they are jointly trained, although with different objectives, is *binary classification*. In fact, the discriminator will output a probability value for each sample, that will be used to classify the samples as one of two classes: *real* or *fake*.

10.1.2 Formalization

Let's try to formalize this idea. Let D_γ be the decoder, or the generator, parametrized by some parameters γ, and Δ_δ be the discriminator. Suppose that we have

- $\mathbf{x}$: a *real* sample from the *real* distribution;
- $\mathbf{x}' = D_\gamma(\mathbf{z})$: a *generated* sample, i.e. the output of the decoder supplied with a latent code $\mathbf{z}$, drawn from some probability distribution defined over the latent space.

Suppose that the distribution of the real data is $p_d(\mathbf{x})$, while the generated data follows a distribution $p_g(\mathbf{x})$. We say that the generator is performing well if it is

$$p_g(\mathbf{x}') \approx p_d(\mathbf{x}). \tag{10.1}$$

On the other hand, the discriminator Δ is performing well if it is

$$\begin{cases} \Delta_\delta(\mathbf{x}) \approx 1 & \overbrace{\mathbf{x} \sim p_d(\mathbf{x})}^{\text{on real data}} \\ \Delta_\delta(\mathbf{x}') \approx 0 & \underbrace{\mathbf{x}' \sim p_g(\mathbf{x}')}_{\text{on fake data}} \end{cases}. \tag{10.2}$$

Let's start from the discriminator. Mathematically, we can define an *objective function* for the discriminator that reflects the general objective defined in eq. (10.2):

$$\max_\delta \underbrace{\mathbb{E}_\mathbf{x} \log \Delta_\delta(\mathbf{x})}_{\text{real data}} + \underbrace{\mathbb{E}_\mathbf{z} \log\left(1 - \Delta_\delta(D_\gamma(\mathbf{z}))\right)}_{\text{fake data}}. \tag{10.3}$$

Let's try to understand what is happening. Recall that the discriminator Δ_δ acts as a *binary classifier*, so it outputs a scalar in $[0, 1]$ that is the *probability that a sample is real*.

- $\mathbb{E}_\mathbf{x} \Delta_\delta(\mathbf{x})$ is the average prediction that the discriminator makes on real data. We want our discriminator to recognize these samples as real, so we want this value to be as close to 1 as possible.
- $\mathbb{E}_\mathbf{z}\left(\Delta_\delta(D_\gamma(\mathbf{z}))\right)$ is the average prediction that the classifier makes on data generated by the decoder D, when supplied with latent codes $\mathbf{z}$. We want the classifier to recognize these samples as fake, so we want this value to be as close to 0 as possible, or, equivalently, we want $\mathbb{E}_\mathbf{z}\left(1 - \Delta_\delta(D_\gamma(\mathbf{z}))\right)$ to be as close to 1 as possible, so that overall we have a maximization problem.

The log has its justification, for now we can just say that since it is a monotonically increasing function, maximizing $\mathbb{E}_{\mathbf{x}}\Delta_\delta(\mathbf{x})$ or $\mathbb{E}_{\mathbf{x}}\log\Delta_\delta(\mathbf{x})$ is the same thing. This is called *success rate*, and an optimal classifier would be a global optimizer to eq. (10.3).
On the other hand, the generator has an opposite objective: it tries to make eq. (10.3) as small as possible. This would mean that it is so good that is able to fool the classifier every time. So, it should find the parameters γ such that when it produces a sample and gives it to the classifier, then its score is as low as possible.
Mathematically this translates to the following objective function:

$$\min_{\gamma}\ \max_{\delta}\ \mathbb{E}_{\mathbf{x}}\log\Delta_\delta(\mathbf{x}) + \mathbb{E}_{\mathbf{z}}\log\left(1-\Delta_\delta(D_\gamma(\mathbf{z}))\right) \tag{10.4}$$

The problem in eq. (10.4) is a minimax problem, and in general these are very difficult problems from the point of view of optimization, but we will see that GANs provide a way to optimize this kind of problems.
In the following we will assume that the generated data follows a distribution $\mathbf{x}\sim p_g$, where p_g is parametrized by γ (the parameters of the generator), and that the real data follows a distribution $\mathbf{x}\sim p_d$.
Let's consider the discriminator: it has to maximize its score, let's call it $J(\cdot)$, given a generator G, over which has no control, so it is a variable for the score:

$$J(G) = \max_{\delta}\mathbb{E}_{\mathbf{x}\sim p_d}\log\Delta_\delta(\mathbf{x}) + \mathbb{E}_{\mathbf{x}\sim p_g}\log\left(1-\Delta_\delta(\mathbf{x})\right). \tag{10.5}$$

Now, let's write explicitly the expectations

$$J(G) = \max_{\delta}\int\left[\log\Delta_\delta(\mathbf{x})p_d(\mathbf{x}) + \log\left(1-\Delta_\delta(\mathbf{x})\right)p_g(\mathbf{x})\right]d\mathbf{x} \tag{10.6}$$

Now we reason pointwise, i.e. we operate outside the integral on an element $\mathbf{x}$ at a time. For any given $\mathbf{x}$, we want to maximize $\Delta_\delta(\mathbf{x}) = a$. Let's rename for simplicity $p_d(\mathbf{x})\equiv p$ and $p_g(\mathbf{x})\equiv q$. Then for a single $\mathbf{x}$ the score is:

$$\max_{a} p\log(a) + q\log(1-a). \tag{10.7}$$

We can see that eq. (10.7) is a concave function, meaning that we can find the maximum just by taking the gradient and equating the gradient to zero:

$$\frac{p}{a} - \frac{q}{1-a} = 0 \tag{10.8}$$

$$a = \frac{p}{p+q} \tag{10.9}$$

We thus have a closed-form solution for the optimal discriminator on a single sample:

$$\Delta_\delta(\mathbf{x}) = \frac{p_d(\mathbf{x})}{p_d(\mathbf{x})+p_g(\mathbf{x})}. \tag{10.10}$$

Since we did not assume anything about the sample, this must hold for any general sample, so we can plug this expression back in eq. (10.5), to get:

$$J(G) = \max_{\delta}\mathbb{E}_{\mathbf{x}\sim p_d}\log\frac{p_d(\mathbf{x})}{p_d(\mathbf{x})+p_g(\mathbf{x})} + \mathbb{E}_{\mathbf{x}\sim p_g}\log\frac{p_g(\mathbf{x})}{p_d(\mathbf{x})+p_g(\mathbf{x})}. \tag{10.11}$$

Let's now define a new distribution, that behaves like a "midpoint distribution" between p_d and p_g, i.e. it is defined point-wise as

$$\rho = \frac{1}{2}p_d + \frac{1}{2}p_g. \tag{10.12}$$

Then, we can rewrite eq. (10.11) as

$$J(G) = \max_{\delta} \frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_d} \log \frac{p_d(\mathbf{x})}{\rho(\mathbf{x})} + \frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_g} \log \frac{p_g(\mathbf{x})}{\rho(\mathbf{x})} + \text{const} \tag{10.13}$$

in which we recognize $\mathbb{E}_{\mathbf{x}\sim p_d} \log \frac{p_d(\mathbf{x})}{\rho(\mathbf{x})}$ and $\mathbb{E}_{\mathbf{x}\sim p_g} \log \frac{p_g(\mathbf{x})}{\rho(\mathbf{x})}$ as Kullback-Leibler divergences, so we write:

$$J(G) = \max_{\delta} \frac{1}{2}KL(p_d\|\rho) + \frac{1}{2}KL(p_g\|\rho) + \text{const}. \tag{10.14}$$

Now, this is the expression of the score of the optimal discriminator, given a generator G. Recall, the generator wants to to minimize the discriminator score, therefore the optimal GAN generator is found by minimizing:

$$\min_{p_g} \frac{1}{2}KL(p_d\|\rho) + \frac{1}{2}KL(p_g\|\rho) + \text{const} \tag{10.15}$$

and since the constant is just a shift in the energy profile of this optimization problem and it does not depend on p_g we can discard it:

$$\min_{p_g} \frac{1}{2}KL(p_d\|\rho) + \frac{1}{2}KL(p_g\|\rho). \tag{10.16}$$

The most notable thing about this expression is that it is the definition of another divergence measure over probability distributions: the *Jensen-Shannon divergence* between p_d and p_g.
Like the KL divergence, it is not actually a distance because it is not symmetric but the interesting thing is that it is "almost a distance". In fact, it satisfies a *defining property* for distance functions, namely the property that if the two distributions p_d, p_g are equal, so they are actually the same distribution, then the Jensen-Shannon divergence is equal to zero and vice versa; in metric spaces this is called identity of indiscernibles.
This, in the GAN setting, means that the globally optimal generator has a distribution exactly equal to the real distribution of the data, and this is exactly what we wanted. By minimizing eq. (10.16) we are pushing p_g to be more and more similar to p_d, so that in the irrealistic hypothesis of finding an optimal generator, they would be exactly equal. In practice, it is *very hard* to train correctly a GAN, since there are numerous problems that can arise. For instance, competition between the two models can very easily tip in favor of one of the two, that effortlessly beats the other in a way that is not informative for the other to improve, and so the training is stuck.

10.2 Adversarial attacks

The idea of GANs is that we are training the generator on data that is produced by an adversary. This is an example of a more general concept called *adversarial training*, in which the data samples that are used for training are called *adversarial examples*.
They are not just useful for training more accurate models, but they can also be used *maliciously*, to fool a model with the sole purpose of making it fail.
The existence of adversarial examples for a model, meaning examples over which the model fails when it is not expected to do so, can be an indicator of poor *robustness* of the model: if we can find such adversarial examples it means that our trained model is not very robust.

Examples Suppose we have a model trained to recognize traffic signs, and that it is able to recognize regular stop signs. Now, on the stop signal in fig. 10.3 was applied some kind of perturbation, in the form of two writings, in a very specific configuration of color and position. The model fails, recognizing the sign as a speed limit sign, instead of a stop sign. This particular adversarial example was found by a generator that has been trained specifically to have the original classifier misclassify the stop sign.

Figure 10.3: An example of malicious application.

In fig. 10.4 we can see two pictures of a school bus. They are the same picture to a human observer, but actually the second is the result of perturbation being applied the first one, in order to get a classifier to misclassify it as something else, an ostrich in this example. Perturbations added to the original sample are usually explicitly optimized to be imperceptible.

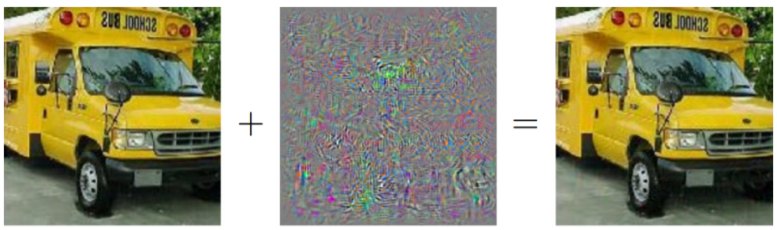
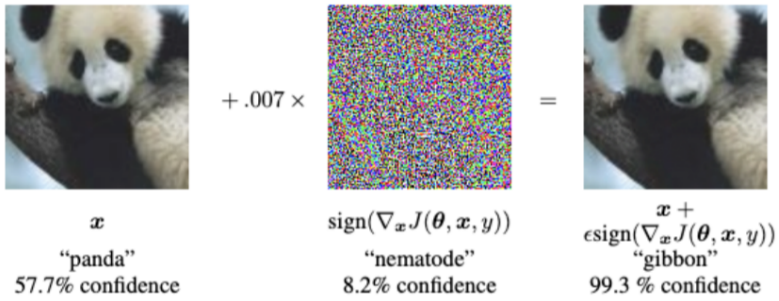


Figure 10.4: Imperceptible adversarial attack.

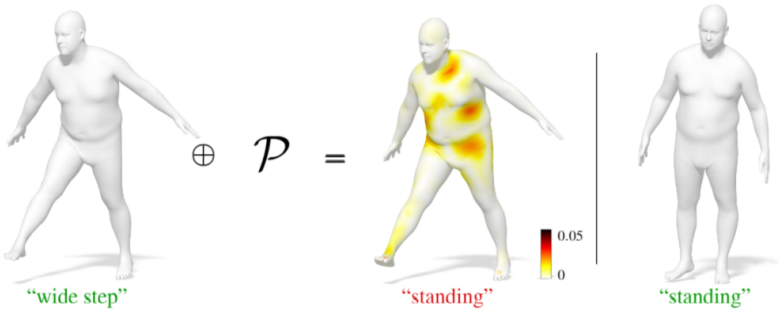
Perception Let's now see how to construct undetectable adversarial examples.
An attack is said to be *undetectable* if it can not be perceived as such. However this is only a loose definition, and we need a *metric* or *measure* that quantifies how good we are at perceiving stuff. This measure should:

- capture the *noticeability* of the attack, i.e. how much the attack is noticeable by a perceiver;
- be *minimizable*, so we can explicitly construct undetectable adversarial examples: if we define the "optimal" adversarial example to be the one that makes this measure as small as possible, then the adversarial example will be as little noticeable as possible.

The choice of such measure will depend on the domain and on the task.



(a) Classification at pixels level.



(b) Classification of pose of a 3D shape.

Figure 10.5: Different perturbations for different tasks.

Types of attacks We can classify attacks based on how much information we have on the model that we are trying to attack.

- **Black-box attack**.
  We are given a trained deep neural network but we do not know anything about the network, we can just give input samples to the network and observe the output.
- **Gray-box attack**.
  We have access to some *partial* information about the model *e.g.* only the features, or the architecture, etc.
- **White-box attack**.
  We have complete access to the network.

We will see *white box attacks* because it has been shown that it is possible to train a *substitute model*, with black box access to the target model that we would like to attack, such that the substitute gives the same input-output pairs. Then we can study attacks the substitute model with a white box approach and then transfer the attack to the original black box target.

Note. When we attack a neural network or learning based model in general, we *cannot change its parameters*. This is not the purpose of the attack: instead, what we want to do is, given a trained network, we want to find a properly crafted sample, possibly undetectable, that the trained network will misclassify. There is no training involved in attacks, since there is no

modifying the network parameters. If that was the case, we could trivially mess with them and make it compute garbage.

Targeted attacks When performing *targeted* attacks, we are given a classifier C and some input sample $\mathbf{x}$ and a *target class* t, meaning the class towards which we want to misclassify. For example, we might want a self-driving car to misclassify a stop sign as a speed limit sign, and in this case this would be a targeted attack, in which the target is the class 'speed limit sign'.
To find an adversarial example for a targeted attack, we solve the following minimization problem:

$$\min_{\mathbf{x}' \in [0,1]^n} \|\mathbf{x} - \mathbf{x}'\|_2^2 \tag{10.17}$$

$$\text{s.t. } C(\mathbf{x}') = t \tag{10.18}$$

so we are trying to look for a new image $\mathbf{x}'$ that is as close as possible to the original one (being imperceptible) and also such that this new image $\mathbf{x}'$ will be classified as the target class.

Note. C is not directly the deep neural network, because the neural network does not output a predicted class, but a probability distribution over all the classes. Usually C is the argmax of the output of the network.

This kind of optimization problem is very difficult since the constraint is highly nonlinear, so we can relax the problem by substituting the constraint with a penalty represented by the cross-entropy loss L:

$$\min_{\mathbf{x}' \in [0,1]^n} \|\mathbf{x} - \mathbf{x}'\|_2^2 + cL(\mathbf{x}', t). \tag{10.19}$$

We still want $\mathbf{x}'$ to be misclassified by the network as t, but we are not imposing a hard constraint any longer, but instead a tradeoff by c.
As c goes to zero, the problem is not really interesting any longer since the $\mathbf{x}'$ that satisfies $\|\mathbf{x} - \mathbf{x}'\|_2^2$ is trivially $\mathbf{x}$.
On the other hand, as c goes to infinity then $\|\mathbf{x} - \mathbf{x}'\|_2^2$ is not considered any longer, and the perturbation applied will be very noticeable, so we want to find a good balance between the two terms, like it is always the case when there is a tradeoff. In this case, one approach is choosing the right value for c via *line search* algorithms.

A more general approach has been proposed. Instead of using the L_2 norm as a distance between adversarial sample and original sample, we can use a more generic notion of distance that depends on the specific problem, and we can explicitly talk about the perturbation δ:

$$\min_{\delta \in [0,1]^n} d(\mathbf{x}, \mathbf{x} + \delta) \tag{10.20}$$

$$\text{s.t. } C(\mathbf{x} + \delta) = t. \tag{10.21}$$

We are optimizing over the space of possible perturbations δ to be added to the original sample. In particular, we are seeking the δ that is the closest possible to the original sample $\mathbf{x}$ in terms of the distance function.
Now, it has been proposed to replace the hard constraint with an inequality constraint:

$$\min_{\delta \in [0,1]^n} d(\mathbf{x}, \mathbf{x} + \delta) \tag{10.22}$$

$$\text{s.t. } f(\mathbf{x} + \delta) \leq 0 \tag{10.23}$$

in which f is a function such that

$$C(\mathbf{x}+\delta) = t \quad \iff \quad f(\mathbf{x}+\delta) \leq 0. \tag{10.24}$$

Several definitions are possible for such a function f.
Then, as the previous approach we can turn the constraint into a penalty:

$$\min_{\delta \in [0,1]^n} d(\mathbf{x}, \mathbf{x}+\delta) + cf(\mathbf{x}+\delta). \tag{10.25}$$

A possible instantiation of this problem is the following:

$$\min_{\delta \in [0,1]^n} \|\delta\|_p + c(\max_{i \neq t} \{F(\mathbf{x}+\delta)_i\} - F(\mathbf{x}+\delta)_t)^+ \tag{10.26}$$

in which:

- The first term is the distance function. It is just the L_p norm of the perturbation δ. If the perturbation has small norm, then the original and adversarial samples will necessarily be close.
- The second term is a possible definition of the function f, actually one that has been shown to work well in practice. Let's see the various terms one by one.
  - $F : \mathbf{x} \to [0,1]^k$ is the neural network that takes as input a value $\mathbf{x}$ and outputs a probability distribution over the k classes.
  - So, $F(\mathbf{x}+\delta)_t$ is the value of the (predicted) probability of the adversarial sample to be of class t.
  - Instead, $\max F(\mathbf{x}+\delta)_i : i \neq t$ is the value of the probability of the "strongest class" (the one the network F has has more confidence predicting) that is not our target class.
  - The $()^+$ notation is just a shorthand notation for $(\cdot)^+ = \max(\cdot, 0)$.

This penalty is then trying to look for the perturbation δ such that when we give $\mathbf{x}+\delta$ to the network, then the probability of class t is the biggest while the probability of all other classes, considered one by one in order of "confidence", is suppressed.

Figure 10.6: Intuition of the definition of f.

Untargeted attacks We perform an *untargeted attack* when we just want a sample to be misclassified with no preference for the target class.
This can be done very efficiently. Suppose we are given an input $\mathbf{x}$ with a ground-truth label ℓ. The network was trained by minimizing the cross-entropy loss $\mathcal{L}(\cdot,\cdot)$, and to misclassify $\mathbf{x}$ it means to increase the loss $\mathcal{L}(\mathbf{x}, \ell)$. Then, we can define our adversarial example as

$$\mathbf{x}' = \mathbf{x} + \underbrace{\epsilon \, sign(\nabla \mathcal{L}(\mathbf{x}, \ell))}_{\text{perturbation}} \tag{10.27}$$

which adds a *perturbation* that maximizes the cost.

Note. The gradient is computed wrt to the sample $\mathbf{x}$, not the model's paramaters. So by increasing $\mathbf{x}$ along the direction of the gradient of the loss *in sample space*, we are effectively increasing the loss by the biggest possible amount.

The parameter ϵ decides how small the perturbation should be. We can iterate eq. (10.27) by performing several "perturbation steps":

$$\mathbf{x'}_{(i)} = \mathbf{x'}_{(i-1)} + \alpha \; sign\left(\nabla\mathcal{L}(\mathbf{x'}_{(i-1)}, \ell)\right), \qquad \mathbf{x'}_{(0)} = \mathbf{x}. \tag{10.28}$$

The parameter $\alpha \ll \epsilon$ is very small, so that we have finer control on the perturbation process.
To prevent the attack from becoming too noticeable, we add a *clipping* operationl. After each perturbation step, we want to make sure that the resulting sample did not go too far away from the original sample. In particular, the *clip* operation projects the sample back into an ϵ-neighborhood of the original sample $\mathbf{x}$.

$$\mathbf{x'}_{(i)} = clip_\epsilon\left(\mathbf{x'}_{(i-1)} + \alpha \; sign\left(\nabla\mathcal{L}(\mathbf{x'}_{(i-1)}, \ell)\right)\right) \tag{10.29}$$

We can notice how the perturbation process might be optimized further by assigning different weights to each component of the sample, instead of adding the same perturbation α to every one of them. However, this method was designed to be efficient, and in fact it can be done with just one backpropagation step, plus some other far less costly operations like clipping, so it is indeed very efficient. Nevertheless, this simple attack can work well in practice.

Vulnerability If we are able to produce adversarial examples then we can use them to train our model, improving the robustness of the attacked learning model.
However, it has been shown that for an arbitrary classifier, no matter how robust we try to make it, there will always exist in theory small adversarial perturbations that make the classifier fail. This means that there is a maximal achievable robustness.
The main observation is that, very often, a trained classifier will learn decision boundaries that are *very close* to the data points. This means that we can move them just by a little bit, and then suddenly they will get to the other side of the boundary which means that they will be misclassified.

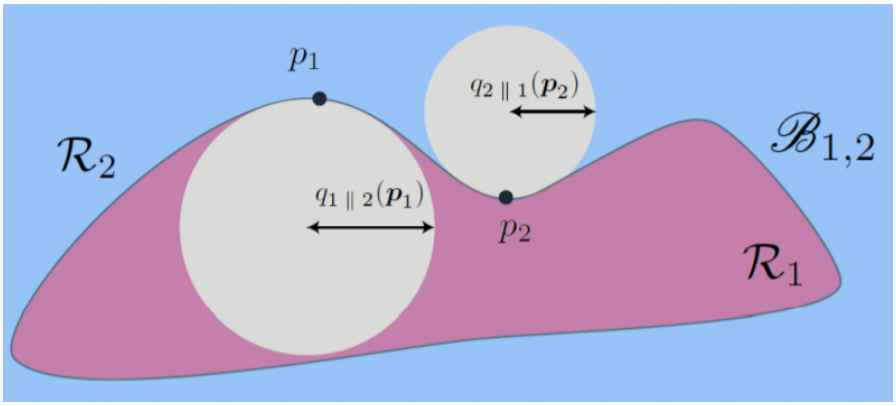


Figure 10.7: (Nonlinear) Decision boundary for an arbitrary learning model.

The success rate of the attacks has been found to be related to the *curvature* of the decision boundary of a neural network classifier. With this mathematical formulation we can quantify the robustness as a function of this curvature.

Universal perturbations So far we have always expressed our attacks in terms of a specific sample $\mathbf{x}$ that we wanted the attacked network to misclassify. However, it would be far mor drastic if we could make a network fail almost on every sample.
In an impactful work of a few years ago, the existence of *universal perturbations* has been shown. They are types of perturbations that, for a given attacked network and a large set of samples $\mathbf{x}$, are adversarial for the entire set, making them *universal*.

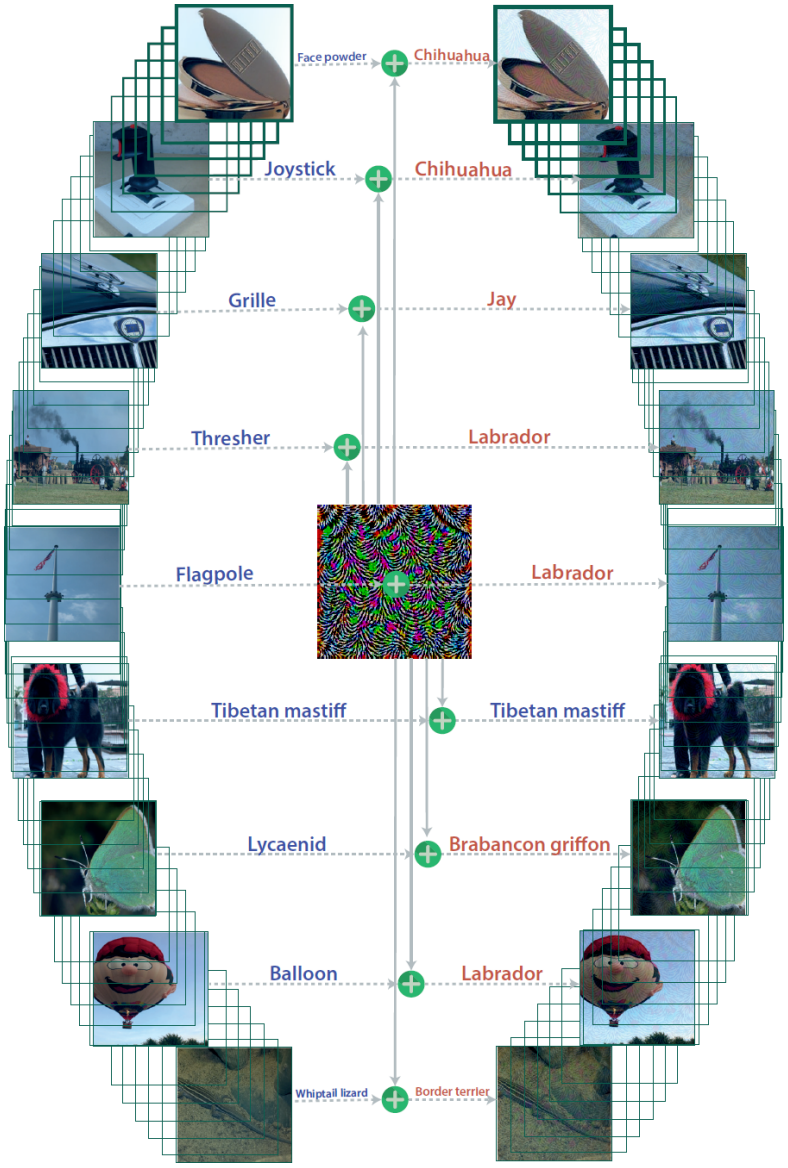


Figure 10.8: The perturbation in the center misclassifies all the images except for the dog.

Closing remarks Adversarial training can also be phrased on non-Euclidean domains like surfaces, graphs, point clouds and other non-Euclidean structures. When we deal with this kind of geometric domains the notion of what is *noticeable* is different than what we had with images, and requires a careful definintion. Also, when defining an attack we can take into consideration the domain itself, beside the data defined over it. For example, we could add or remove edges or, if we were dealing with a point cloud, we could add or move points in space.
There is a whole branch of adversarial machine learning that deals with *adversarial defense*: methods and techniques to shield learning models from adversarial attacks. Adversarial training is one such method.

Appendix A

Linear Algebra

Linear algebra is the study of *linear maps* on finite dimensional *vector spaces*.

A.1 Vector spaces

The motivation for the definition of a vector space comes from the classical properties of addition and scalar multiplication. A *vector space* V over a field F is a set equipped with two operations, $+ : V \to V$ and $\cdot : F \times V \to V$, often referred to as *addition* and *scalar multiplication* respectively, that satisfy the following properties:

- **commutativity**: $u + v = v + u$ for all $u, v \in V$;
- **associativity**: $(u + v) + w = u + (v + w)$ and $(ab)v = a(bv)$ for all $u, v, w \in V$ and all $a, b \in \mathbb{R}$;
- the set is closed wrt to the two operations, so $u + v \in V$ and $av \in V$: "what happens in Vegas, stays in Vegas";
- **additive identity**: there exists an element $0 \in V$ such that $v + 0 = v$ for all $v \in V$
- **additive inverse**: for every $v \in V$, there exists $w \in V$ such that $v + w = 0$
- **multiplicative identity**: $1v = v$ for all $v \in V$
- **distributive properties**: $a(u + v) = au + av$ and $(a + b)v = av + bv$ for all $a, b \in \mathbb{R}$ and all $u, v \in V$

$\mathbb{R}^n$ is defined as the set of all n-long sequences of numbers in $\mathbb{R}$:

$$\mathbb{R}^n = \{(x_1, \ldots, x_n) : x_j \in \mathbb{R} \text{ for } j = 1, 2, \ldots, n\}$$

Addition and scalar multiplication are defined as expected:

$$(x_1, x_2, \ldots, x_n) + (y_1, y_2, \ldots, y_n) = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$$
$$\lambda(x_1, x_2, \ldots, x_n) = (\lambda x_1, \lambda x_2, \ldots, \lambda x_n)$$

While the additive identity can be defined as:

$$0 = (0, \dots, 0)$$

With these definitions, $\mathbb{R}^n$ is a vector space, usually defined over the scalar field $\mathbb{R}$.
Consider the set of all functions $f : [0, 1] \to \mathbb{R}$ with the standard definitions for sum and scalar product:

$$(f + g)(x) = f(x) + g(x)$$
$$(\lambda f)(x) = \lambda f(x)$$

for all $x \in [0, 1]$ and $\lambda \in \mathbb{R}$, with additive identity and inverse defined as:

$$0(x) = 0$$
$$(-f)(x) = -f(x)$$

for all $x \in [0, 1]$. The above forms a vector space. In fact, *any* set of functions $f : S \to \mathbb{R}$ with $S \neq \emptyset$ (Q: why?) and the definitions above forms a vector space.
Elements of a vector space (called *vectors*) are not necessarily lists. A vector space is an *abstract* entity whose elements might be lists, functions, or weird objects. Surfaces do not form a vector space, as the sum of two points over a surface is not defined. Surfaces can be studied using *differential geometry*, which is a mathematical discipline that uses the techniques of differential calculus, integral calculus, linear algebra and multilinear algebra to study problems in geometry; we'll need it for studying the *manifold hypothesis* and *geometric deep learning*.

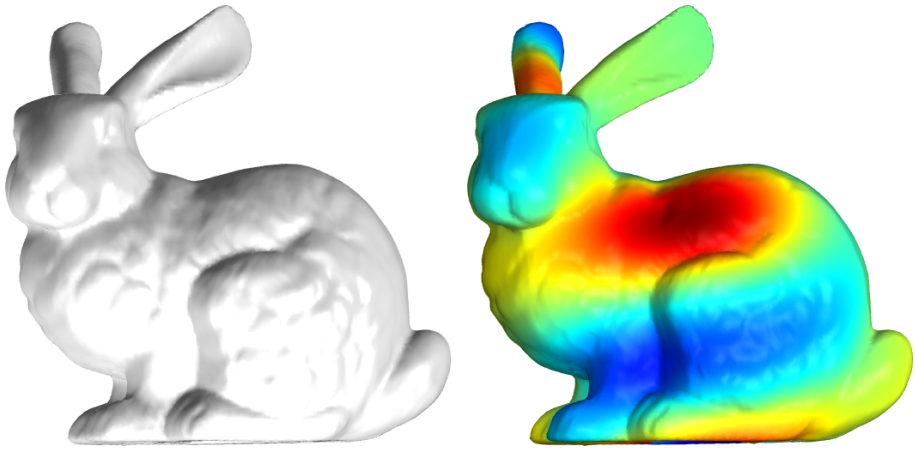


Figure A.1: An example of surface.

We can still use linear algebra to manipulate *functions on surfaces*.
A subset $U \subset V$ is a *subspace* of V if it is a vector space (using the same operations defined for V). In particular:

- $0 \in U$
- $u, v \in U$ implies $u + v \in U$

- $u \in U$ implies $\alpha u \in U$ for any $\alpha \in \mathbb{R}$

Examples:

- $\{(x_1, x_2, 0) : x_1, x_2 \in \mathbb{R}\}$ is a subspace of $\mathbb{R}^3$
- The set of *piecewise-linear functions* on a graph $G = (V, E)$ is a subspace of all functions $f : V \to \mathbb{R}$

A.1.1 Basis

A *basis* of V is a collection of vectors in V that is *linearly independent* and *spans* V

- $\text{span}(v_1, \ldots, v_n) = \{a_1 v_1 + \cdots + a_n v_n : a_1, \ldots, a_n \in \mathbb{R}\}$
- $v_1, \ldots, v_n \in V$ are *linearly independent* if and only if each $v \in \text{span}(v_1, \ldots, v_n)$ has only one representation as a linear combination of $v_1, \ldots, v_n$

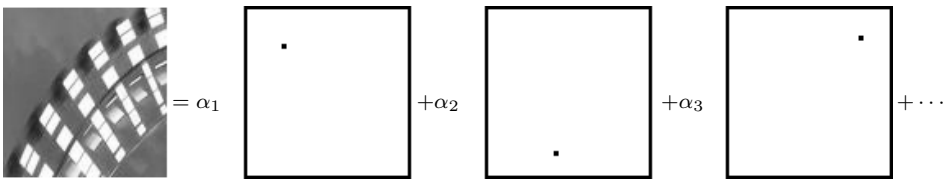
So every vector $v \in V$ can be expressed *uniquely* as a linear combination

$$v = \sum_{i=1}^{n} \alpha_i v_i$$

You can think of a basis as the minimal set of vectors that generates the entire space.

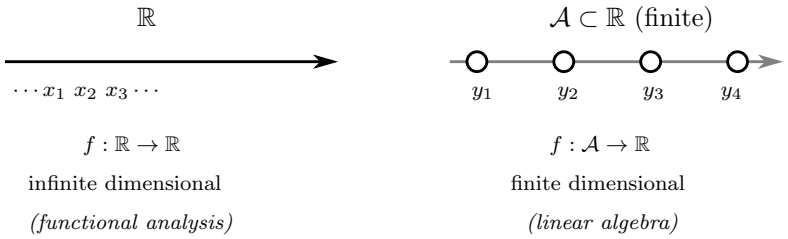
- $(1, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, \ldots, 0, 1)$ is a basis of $\mathbb{R}^n$ called the *standard basis*; its vectors are called the *indicator vectors*. In deep learning, also called *one-hot* representation.
- $(1, 2), (3, 5.07)$ is a basis of $\mathbb{R}^2$
- $$f_1(x) = \begin{cases} 1 & \text{if } x = x_1 \\ 0 & \text{else} \end{cases}$$
  $$f_2(x) = \begin{cases} 1 & \text{if } x = x_2 \\ 0 & \text{else} \end{cases}$$
  $$\vdots$$
  is the standard basis for the set of functions $f : \mathbb{R} \to \mathbb{R}$; the basis vectors are also called *indicator functions*

An image expressed in the *standard basis*:



The same image, expressed in terms of a *nonlinear* map σ:

The image is **not** in the span of the three features.
A vector space may have different bases; any two bases have the *same number of vectors* The *dimension* of a (finite-dimensional) vector space is the length of any basis of the vector space.
Note: Even though function spaces are *not* necessarily finite dimensional (Q: why?), with digital data they usually are, since we deal with finite discrete domains (images, graphs, text, etc.).



A.2 Linear maps

A *linear map* from V to W is a function $T : V \to W$ with the properties:

- **additivity**: $T(u + v) = Tu + Tv$ for all $u, v \in V$
- **homogeneity**: $T(\lambda v) = \lambda(Tv)$ for all $\lambda \in \mathbb{R}$ and all $v \in V$

Examples:

- identity $I : V \to V$, defined as $Iv = v$
- differentiation $D : \mathcal{F}(\mathbb{R}) \to \mathcal{F}(\mathbb{R})$, defined as $Df = f'$
- integration $T : \mathcal{F}(\mathbb{R}) \to \mathbb{R}$, defined as $Tf = \int_0^1 f(x)dx$
- from $\mathbb{R}^3$ to $\mathbb{R}^2$, defined as

$$T(x, y, z) = (2x - y + 3z, 7x + 5y - 6z)$$

- from $\mathbb{R}^n$ to $\mathbb{R}^m$, defined as

$$T(x_1, \ldots, x_n) = (A_{1,1}x_1 + \cdots + A_{1,n}x_n, \ldots, A_{m,1}x_1 + \cdots + A_{m,n}x_n)$$

- equation of a line

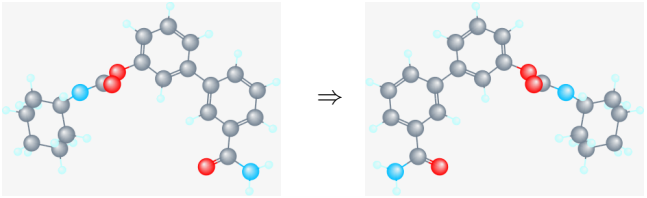
$$y = ax + b$$

- equations such as the following, note that this is linear with respect to $sin(x)$

$$y = z\sin(x) + z^2\sin(x)$$

- *Reflection* operation on an image:

$$T : \mathbb{R}^2 \to \mathbb{R}^2, \quad T(x, y) = (-x, y)$$



Linear maps $T : V \to W$ form a *vector space*, with addition and multiplication defined as:

$$(S + T)(v) = Sv + Tv$$
$$(\lambda T)(v) = \lambda(Tv)$$

In addition, we also have a useful definition of *product* between linear maps. This is kind of a special situation, since multiplying vectors doesn't necessarily make sense in general.
If $T : U \to V$ and $S : V \to W$, their product $ST : U \to W$ is defined by

$$(ST)(u) = S(Tu)$$

In other words, ST is just the usual composition $S \circ T$ of two functions

- **associativity**: $(T_1T_2)T_3 = T_1(T_2T_3)$
- **identity**: $TI = IT = T$
- **distributive properties**: $(S_1 + S_2)T = S_1T + S_2T$ and $S(T_1 + T_2) = ST_1 + ST_2$

Keep in mind that composition of linear maps *is not commutative*, i.e.

$$ST \neq TS$$

in general (although there are special cases).
Example: Take $Sf = f'$ and $(Tf)(x) = x^2 f(x)$.

A.2.1 Matrices

Consider a linear map $T : V \to W$, a basis $v_1, \ldots, v_n \in V$ and a basis $w_1, \ldots, w_m \in W$. The *matrix* of T in these bases is the $m \times n$ array of values in $\mathbb{R}$

$$\mathbf{T} = \begin{pmatrix} T_{1,1} & \cdots & T_{1,n} \\ \vdots & & \vdots \\ T_{m,1} & \cdots & T_{m,n} \end{pmatrix}$$

whose entries $T_{i,j}$ are defined by

$$Tv_j = T_{1,j}w_1 + \cdots + T_{m,j}w_m$$

Hence each column of $\mathbf{T}$ contains the linear combination coefficients for the image via T of a basis vector from V

In other words, the matrix encodes how basis vectors are mapped, and this is enough to map all other vectors in their span, since:

$$Tv = T(\sum_j \alpha_j v_j) = \sum_j T(\alpha_j v_j) = \sum_j \alpha_j T v_j$$

The matrix is a *representation* for a linear map, and it *depends on the choice of bases*. Suppose $v \in V$ is an arbitrary vector, while $v_1, \dots, v_n$ is a basis of V. The matrix of v wrt this basis is the $n \times 1$ matrix:

$$\mathbf{v} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$

so that

$$v = c_1 v_1 + \cdots c_n v_n$$

Once again, we see that the matrix *depends on the choice of basis* for V

- **addition**: the matrix of $S + T$ can be obtained by summing the matrices of S and T; this only makes sense if the *same bases* are used for S, T, and $S + T$
- **scalar multiplication**: given $\lambda \in \mathbb{R}$, the matrix for λT is given by λ times the matrix of T

In fact, we have just shown that *matrices form a vector space* (Q1: what is the additive identity?) (Q2: what is the vector space dimension?)
We call $\mathbb{R}^{m \times n}$ the vector space of all $m \times n$ matrices with values in $\mathbb{R}$

- **product**: the matrix for ST can be computed by the *matrix product* between **S** and **T**; in fact, the matrix product is defined precisely to make this work

  Q3: is matrix product commutative?

  Q4: do we need the same bases for $S : U \to V$ and $T : V \to W$?

Consider a linear map $T : V \to W$, a basis $v_1, \dots, v_n \in V$ and a basis $w_1, \dots, w_m \in W$.
From the definition of matrix product, one can show that it operates on a vector matrix as expected:

$$\mathbf{Tv} = \mathbf{w} \quad \Leftrightarrow \quad Tv = w$$

where **Tv** is the matrix product of **T** and **v**, while Tv simply denotes the function evaluation $T(v)$
Remember: $\mathbf{T}, \mathbf{v}, \mathbf{w}$ must follow a coherent choice of bases in order for the above to make sense. **v** can not be expressed in basis $(\tilde{v}_1, \dots, \tilde{v}_n)$ if **T** only knows how to map basis vectors $(v_1, \dots, v_n)$.

$$Tv_j = T_{1,j} w_1 + \cdots + T_{m,j} w_m$$

$$\underbrace{\begin{pmatrix} T_{1,1} & \cdots & T_{1,n} \\ \vdots & & \vdots \\ T_{m,1} & \cdots & T_{m,n} \end{pmatrix}}_{\mathbf{T}} \underbrace{\begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}}_{\mathbf{c}} = \sum_{j=1}^{n} c_j \underbrace{\begin{pmatrix} T_{1,j} \\ \vdots \\ T_{m,j} \end{pmatrix}}_{\mathrm{Tv_j\ wrt\ (w_1,\dots,w_m)}}$$

Because recall that, for bases $v_1, \ldots, v_n \in V$ and $w_1, \ldots, w_m \in W$:

$$Tv_j = T_{1,j}w_1 + \cdots + T_{m,j}w_m$$

We see then that vector $c = \sum_j c_j v_j$ is mapped to $Tc = \sum_j c_j T v_j$.
In other words, matrix product is behaving as expected.

A.3 Matrix meta-mechanics

The definition of matrix product gives rise to some alternative viewpoints that are often useful for practical manipulation of matrices.

Transpose and inverse

Definition A.1 (Symmetric matrix)**.** A matrix A is symmetric if $A = A^\top$.

If the matrix A is a product $A = BC$, the transpose applies as follows

$$(BC)^\top = C^\top B^\top \tag{A.1}$$

and the same holds for the inverse

$$(BC)^{-1} = C^{-1}B^{-1} \tag{A.2}$$

Definition A.2 (Orthogonal Matrix)**.** A matrix A is orthogonal if its columns and rows are orthogonal unit vectors (orthonormal vectors). Equivalently, a matrix A is orthogonal if

$$A^{-1} = A^\top \tag{A.3}$$

Thus $A^\top A = I$ whenever A is orthogonal.

Claim A.1. *If A is orthogonal then $A^\top A = I \iff AA^\top = I$.*

Proof.

$$A^\top A = I \qquad \text{(by multiplying both sides for } A\text{)}$$
$$AA^\top A = AI \tag{A.4}$$

And this implies that

$$AA^\top = I \tag{A.5}$$

□

Products The matrix-vector product is as follows

$$\mathbf{Xy} = \begin{pmatrix} | & | & \\ \mathbf{x_1} & \mathbf{x_2} & \cdots \\ | & | & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \end{pmatrix} = y_1 \begin{pmatrix} | \\ \mathbf{x_1} \\ | \end{pmatrix} + y_2 \begin{pmatrix} | \\ \mathbf{x_2} \\ | \end{pmatrix} + \ldots \tag{A.6}$$

while the vector-matrix product is just a transposed version of the above

$$\mathbf{z}^\top \mathbf{A} = (\mathbf{A}^\top \mathbf{z})^\top \tag{A.7}$$

Matrix-matrix product:

$$\mathbf{XY} = \begin{pmatrix} - \mathbf{x}_1^\top - \\ - \mathbf{x}_2^\top - \\ \vdots \end{pmatrix} \begin{pmatrix} | & | \\ \mathbf{y}_1 & \mathbf{y}_2 \cdots \\ | & | \end{pmatrix} = \begin{pmatrix} | & | \\ \mathbf{x}\mathbf{y}_1 & \mathbf{x}\mathbf{y}_2 \cdots \\ | & | \end{pmatrix} \tag{A.8}$$

$$= \begin{pmatrix} - \mathbf{x}_1^\top \mathbf{Y} - \\ - \mathbf{x}_2^\top \mathbf{Y} - \\ \vdots \end{pmatrix} \tag{A.9}$$

A.4 Other properties

The vector projection of a vector $\mathbf{a}$ on (or onto) a nonzero vector $\mathbf{b}$ is the orthogonal projection of $\mathbf{a}$ onto a straight line parallel to $\mathbf{b}$. It is a vector parallel to $\mathbf{b}$, defined as: $\mathbf{a}_1 = a_1\hat{\mathbf{b}}$, where a_1 is a scalar, called the scalar projection of $\mathbf{a}$ onto $\mathbf{b}$, and $\hat{\mathbf{b}}$ is the unit vector in the direction of $\mathbf{b}$. In turn, the scalar projection is defined as: $a_1 = \|\mathbf{a}\| \cos\theta = \mathbf{a} \cdot \hat{\mathbf{b}} = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{b}\|}$ where the operator $\cdot$ denotes a dot product, $\|\mathbf{a}\|$ is the length of $\mathbf{a}$, and θ is the angle between $\mathbf{a}$ and $\mathbf{b}$. The scalar projection is equal to the length of the vector projection, with a minus sign if the direction of the projection is opposite to the direction of $\mathbf{b}$.

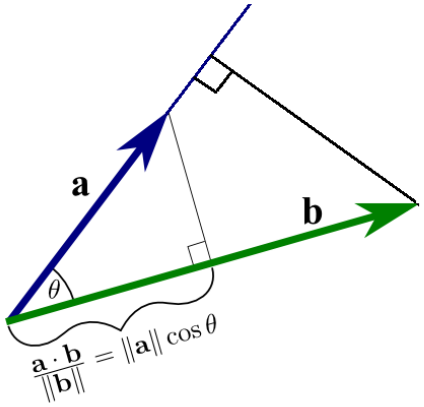


Figure A.2: Projection of $\mathbf{a}$ on $\mathbf{b}$

Appendix B

Information Theory

B.1 Entropy

Intuition In information theory, the *entropy* of a random variable is the average level of "information", or dually "uncertainty", that is inherent in the variable's possible outcomes.
Let's look at one such outcome, let's call it the event E, and say that it will occur with probability $p(E)$.
The intuition is that if the event E is very likely to happen, then knowing that it will happen does not bring any interesting information. On the contrary, what is truly informative is knowing that something that happens very infrequently will indeed happen.
Therefore, the *information content* (or *surprisal*) carried by knowing that the event E will happen can be quantified as a function that decreases with $p(E)$. In particular, this function is defined as:

$$I(E) \triangleq -\log p(E) = \log \frac{1}{p(E)}. \tag{B.1}$$

A random variable has a probability distribution defined over all the events that it encodes. Then, we can compute the average information inherent in the variable as the weighted average of the information carried by each one of its events, weighted by the probability that it will actually happen.
A random variable in which all the outcomes are equally likely has high entropy, since there is maximum uncertainty about its outcome. A random variable in which there are certain outcomes that are much more likely than others has low entropy, since there is less uncertainty about its outcome.
A random variable that has *more* outcomes than another has *higher* entropy, since there is more uncertainty about its outcome.

Definition Let X be a random variable, with possible values $\{x_1, \ldots, x_n\}$. Let $P(X)$ be the probability mass function defining a probability distribution over all the possible values of X. Then, we call *entropy* the average information represented in the distribution:

$$H(X) \triangleq -\sum_{i=1}^{n} p(x_i) \log p(x_i), \tag{B.2}$$

where $p(x_i) \equiv P(X = x_i)$.

B.2 Kullback-Leibler divergence

Intuition Let's consider two distributions of probability p and q. Usually, p represents the data, while q a model, or in general an approximation of p, and we want to know how good of an approximation this is.
Then the Kullback-Leibler divergence is interpreted as the average loss of information content that we have when representing samples of p (the data) using an *optimal code* (something that does not introduce additional uncertainty beside the one intrinsic to the distribution) for q (the model) instead of an optimal code for p.

Definition Let p and q be two probability distributions defined over the same space X.
Then we call Kullback-Leibler divergence the measure

$$KL(p\|q) \triangleq \sum_{x\in X} p(x)\log\left(\frac{p(x)}{q(x)}\right) \tag{B.3}$$

which is equivalent to

$$KL(p\|q) = -\sum_{x\in X} p(x)\log\left(\frac{q(x)}{p(x)}\right). \tag{B.4}$$

Properties The KL-divergence:

- is *not* symmetric.
- is *not* a distance, since it is not symmetric.
- is always non-negative.
- can be expressed as an expectation.

$$\begin{aligned} KL(P\|Q) &= \sum_{x\in X} p(x)\log\left(\frac{p(x)}{q(x)}\right) \\ &= \mathbb{E}_{x\sim p(x)}\log\left(\frac{p(x)}{q(x)}\right). \end{aligned} \tag{B.5}$$

- measures the dissimilarity of two distributions in terms of their entropy.

$$KL(p\|q) \approx H(q) - H(p) \tag{B.6}$$

In fact, Substituting the definition of entropy in eq. (B.6) we go back to the definition.

$$KL(p\|q) \approx -\sum_x q(x)\log\ q(x) + \sum_x p(x)\log\ p(x) \tag{B.7}$$

$$KL(p\|q) = -\sum_x p(x)\log\ q(x) + \sum_x p(x)\log\ p(x) \tag{B.8}$$

$$KL(p\|q) = \sum_x p(x)\log\frac{p(x)}{q(x)} \tag{B.9}$$

Appendix C

Fourier Analysis

C.1 Signals

Signal processing is the field of engineering that studies *signals*, how to analyse, modify and synthesise them.
What is a signal? Its definition is not a strict one, on the contrary, it is one of the loosest definition one may find. Intuitively, a signal exists as a *carrier of information*, and this information can be of various nature, hence to capture a large spectrum of phenomena the definition is kept loose.
We may define a signal as *any variable physical quantity that conveys information about a phenomenon*. With this definition, the formalization of a signal is one of a *mathematical function* of one or more variables. The independent variables in the physical world may be time, or space, or both. The dependent variable describes the information of interest.
For example, a song can be thought of as a signal

$$x(t) : \mathbb{R} \to \mathbb{R} \tag{C.1}$$

in which the independent variable t is time, so that the signal tells us the intensity of sound at that moment.
Also an *image* may be though of as a signal, since it respects the definition we gave: the information of an image is the intensity values of all the pixels that compose it. So, it can be described as a signal

$$f(x_1, x_2) : \mathbb{R}^2 \to \mathbb{R} \tag{C.2}$$

in which the independent variables are *spatial*, so that the signal tells us the *intensity* of a generic point $\mathbf{x} = (x_1, x_2)^\top$ of the image.

C.2 Fourier series

We like simple things. Signals, as functions, can be very complicated functions, very difficult to process. It would be nice to be able to *decompose* a complicated signal as combination of several *elementary* signals, each one of known characteristics and hence easy to process, and so be able to process the original signal considered as a *superposition* of these elementary signals.
What would be an elementary signal? The most elementary of signals is the *constant* signal, but as you can imagine we cannot describe arbitrary signals as combination of constant signals. As it turns out, in signal processing we like *sinusoids*, or *sine waves*, since they are easily described.

A sine wave is a function

$$x(t) = A\sin(2\pi ft + \phi), \tag{C.3}$$

here described as a function of time t.
A sine wave is *periodic*, i.e. it is

$$x(t + T_0) = x(t) \tag{C.4}$$

so the signal repeats itself after every cycle of width T_0, and it suffices to describe the signal in this window to be able to describe the whole signal. Most notably, we can *completely* characterize a sine wave in terms of just three quantities: *amplitude* A, *frequency* f_0 and *phase* ϕ.

- the amplitude tells us the maximum (and minimum) value that the sine wave can take;
- the frequency tells us the number of cycles that occur each second of time (or in each unit of the independent variable, in general);
- the phase tells us the *displacement* of the sine wave, i.e. where in its cycle the oscillation is at the initial instant $t = 0$. A sine wave with $\phi = 0$ starts at the origin at $t = 0$. A sine wave with $\phi = \pi/2$ starts at A at the origin, and actually becomes a cosine wave.

Given these three parameters, a sine wave is completely determined. Therefore, it would be nice to be able to describe arbitrarily complicated signals as combinations of sine waves, since that would mean combinations of just these three pieces of information for each member of the combination.
The *Fourier series* allows us to describe any *periodic* signal as a composition of sine waves. These sine waves are *harmonically* related to each other, meaning that their frequency is not random, but is a multiple of a *fundamental frequency* f_0, so that their combination is able to keep the periodic nature of the original signal. In fact, if a time signal $x(t)$ is periodic of period T_0, it will repeat itself $f_0 = \frac{1}{T_0}$ times in each unit of time.
The k-th sinusoid will have frequency $2\pi kf_0$, and will be called k-th *harmonic oscillation*, or simply k-th *harmonic*.
Now, given a certain harmonic, we have to specify what is its amplitude and its phase. We could do so with two numbers, but actually *complex numbers* can be described in *polar coordinates* with exactly this two quantities, and furthermore thanks to the *Euler formula*

$$e^{i\theta} = cos(\theta) + i \cdot sin(\theta) \tag{C.5}$$

we can express a sine wave as a combination of complex numbers:

$$A\sin(2\pi ft + \phi) = A\frac{e^{(2\pi ft+\phi)i} - e^{-(2\pi ft+\phi)i}}{2i} \tag{C.6}$$

where i is the imaginary unit.
Actually, since it is:

$$A\cos(2\pi ft + \phi) = A\frac{e^{(2\pi ft+\phi)i} + e^{-(2\pi ft+\phi)i}}{2} \tag{C.7}$$

we prefer to use *cosine waves*, since as we have seen one can obtain the other by a shift in phase of $\frac{\pi}{2}$.
So the k-th harmonic will be a function:

$$\begin{aligned} A_k\cos(2\pi kf_0t + \phi_k) &= A_k\frac{e^{(2\pi kf_0t+\phi_k)i} + e^{-(2\pi kf_0t+\phi_k)i}}{2} \\ &= \underbrace{A_k}_{\text{amplitude}}\overbrace{e^{i\phi_k}}^{\text{phase}}\underbrace{e^{2\pi kf_0t}}_{\text{frequency}} + A_ke^{-i\phi_k}e^{-2\pi kf_0t}. \end{aligned} \tag{C.8}$$

so that we can express a signal $x(t)$ as the Fourier series:

$$\begin{aligned} x(t) &= \sum_{k=0}^{\infty} A_k e^{i\phi_k} e^{2\pi k f_0 t} + \sum_{k=0}^{\infty} A_k e^{-i\phi_k} e^{-2\pi k f_0 t} \\ &= \sum_{k=0}^{\infty} A_k e^{i\phi_k} e^{2\pi k f_0 t} + \sum_{k=-\infty}^{-1} A_{-k} e^{-i\phi_{-k}} e^{-2\pi k f_0 t} \\ &= \sum_{k=-\infty}^{\infty} X_k e^{2\pi k f_0 t} \end{aligned} \tag{C.9}$$

in which we have defined

$$X_k \triangleq \begin{cases} A_k e^{i\phi_k} & k = 1, 2, \dots \\ A_{-k} e^{-i\phi_{-k}} & k = \dots, -2, -1. \end{cases} \tag{C.10}$$

- X_k is a complex number called *Fourier coefficient*, whose magnitude $|X_k|$ encodes the amplitude of the k-th harmonic and whose phase $\phi(X_k)$ encodes the phase of the k-th harmonic.
- $e^{2\pi k f_0 t i}$ is a member of the *Fourier basis*.

We are expressing the periodic signal $x(t)$ as the infinite discrete combination of the Fourier basis through the complex coefficients X_k. Without showing the derivation, we can compute X_k from the original signal $x(t)$ as:

$$X_k = \frac{1}{T_0} \int_{-T_0/2}^{T_0/2} x(t) e^{-2\pi k f_0 t i} dt. \tag{C.11}$$

So to summarize for a periodic signal $x(t)$ we have an equation of *analysis*, that allows us to *decompose* the signal in the *time domain* as the combinations of elementary signals (represented as Fourier coefficients in the complex domain, also called Fourier or *frequency domain*), and an equation of *synthesis*, that allows us to *reconstruct* the signal given its representation in the *frequency domain*:

$$x(t) = \sum_{-\infty}^{\infty} X_k e^{2\pi k f_0 t i}, \qquad X_k = \frac{1}{T_0} \int_{-T_0/2}^{T_0/2} x(t) e^{-2\pi k f_0 t i} dt. \tag{C.12}$$

In the first relation we need to know the fundamental frequency f_0 and the *sequence* of Fourier coefficients $X_k = \{X_0, \dots, X_k, \dots\}$; in the second relation we need to know the period T_0 and the behavior in time $x(t)$ of the signal. The two relations allow to establish a 1-to-1 correspondance

$$x(t) \iff X_k \tag{C.13}$$

such that knowing the signal in the *time domain* $x(t)$ or knowing the signal in the *frequency domain* as the sequence X_k has the same information content.

C.3 Fourier transform

Recall that we can decompose a signal $x(t)$ as a Fourier series only if the signal is *periodic*. Can we say something about arbitrary signals $x(t)$? It turns out that we can, through the *Fourier transform*.

We can still decompose such signals as a proper superposition of elementary (cosinusoidal) signals, but this superposition will be different. We can think of an *aperiodic* signal as a signal whose period is "infinite", meaning that it never repeats itself. In the limit of $T_0 \to \infty$, all the frequencies kf_0 of the harmonics go to zero, they become *infinitesimal* frequencies.
If we think of X_k as the sampling of a continuous function $X(f)$ of frequency, discretely sampled at the harmonic frequencies kf_0:

$$X(kf_0) \triangleq T_0 X_k = \frac{1}{f_0} X_k \tag{C.14}$$

then for a periodic signal we have

$$\begin{aligned} x(t) &= \sum_{k=-\infty}^{\infty} X_k e^{2\pi k f_0 t i} \\ &= \sum_{k=-\infty}^{\infty} X(kf_0) e^{2\pi k f_0 t i} \cdot f_0. \end{aligned} \tag{C.15}$$

As the signal becomes aperiodic, $T_0 \to \infty$ and $f_0 \to 0$, so that both the increment f_0 between two sampled points $kf_0, (k+1)f_0$ becomes infinitesimal, and we have an infinite sum of samples at infinitesimal distance:

$$\begin{aligned} x(t) &= \lim_{f_0 \to 0} \sum_{k=-\infty}^{\infty} X(kf_0) e^{2\pi k f_0 t} \cdot f_0 \\ &= \int_{-\infty}^{\infty} X(f) e^{2\pi f t i} df \end{aligned} \tag{C.16}$$

By definition, this is an integral, the *Fourier integral*.
We have expressed the signal $x(t)$ as a infinite *continuous* superposition of elementary signals. Every frequency f (varying continuously) will be needed to make up the signal, not just the discrete sampled frequencies kf_0. So $X(f)$ is a *complex function* of the *continuous variable* f. By taking a similar limit we get

$$\begin{aligned} X(f) &= \lim_{f_0 \to 0} \frac{1}{T_0} \int_{-T_0/2}^{T_0/2} x(t) e^{-2\pi k f_0 t i} dt \\ &= \int_{-\infty}^{\infty} x(t) e^{-2\pi f t i}. \end{aligned} \tag{C.17}$$

The same correspondance that existed for periodic signals still exist:

$$x(t) \iff X(f) \tag{C.18}$$

so knowing the signal in the *time domain* $x(t)$ or knowing the signal in the *frequency domain* $X(f)$ has the same information content.
Since now decomposing a signal means obtaining a function from another function, we can think of the operation in eq. (C.17) as a *transform*, i.e. an operator that takes a function and suitably transforms it to obtain another function. $X(f)$ is also called the *Fourier transform* $\mathcal{F}\{\cdot\}$ of the signal $x(t)$, from the time domain to the frequency domain.
For the same reason we can think of obtaining the original signal $x(t)$ from $X(f)$ as an *inverse Fourier transform* $\mathcal{F}^{-1}\{\cdot\}$ of $X(f)$, from the frequency domain to the time domain.

The equations of *analysis* and *synthesis* now are:

$$x(t) = \underbrace{\mathcal{F}^{-1}\{X(f)\} = \int_{-\infty}^{\infty} X(f)e^{2\pi ft i}}_{\text{Inverse Fourier transform}}, \qquad X(f) = \underbrace{\mathcal{F}\{x(t)\} = \int_{-\infty}^{\infty} x(t)e^{-2\pi ft i}}_{\text{Fourier transform}}. \tag{C.19}$$

Note. Often the Fourier transform of a signal $x(t)$ is also represented as $\hat{x}(f)$, and often also the dependence on the independent variable (in both domains) is omitted:

$$\mathcal{F}\{x\} = \hat{x}, \qquad \mathcal{F}^{-1}\{\hat{x}\} = x. \tag{C.20}$$

C.4 Properties

List of properties, without derivation.

- **Linearity.**
  The Fourier transform is *linear*, meaning

  $$\mathcal{F}\{\alpha x(t) + \beta y(t)\} = \alpha\mathcal{F}\{x(t)\} + \beta\mathcal{F}\{y(t)\}. \tag{C.21}$$

  This property is a direct consequence of the linearity of the integral.

- **Convolution theorem.**
  The Fourier transform *diagonalizes* convolution, meaning that performing the convolution of two signals $x \star y$ in the time domain is equivalent to a simple *multiplication* of the their Fourier transforms in the Fourier domain. By definition of the convolution operator it is:

  $$z(t) = (x \star y)(t) = \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau, \tag{C.22}$$

  but for the convolution theorem, it also is:

  $$\begin{aligned} \hat{z}(f) &= \mathcal{F}\{z(t)\} = \mathcal{F}\{(x \star y)(t)\} \\ &= \dots \\ &= \hat{x}(f)\hat{y}(f). \end{aligned} \tag{C.23}$$

  So, we can perform convolution by transforming the two signals in the Fourier domain, convoluting by a simple multiplication, and then inverse transforming them back to the time domain.

  $$\begin{aligned} (x \star y)(t) &= \mathcal{F}^{-1}\{\mathcal{F}\{(x \star y)(t)\}\} \\ &= \mathcal{F}^{-1}\{\hat{x}(f)\hat{y}(f)\}. \end{aligned} \tag{C.24}$$

Appendix D

Spectral Graph Theory

Let $\mathbf{A}$ be the adjacency matrix of a simple graph G. $\mathbf{A}$ is a real-symmetric matrix, and thus has n real eigenvalues and its n real eigenvectors form an orthonormal basis.
Let $\{\lambda_1, \ldots, \lambda_i, \ldots, \lambda_r\}$ be the set of *distinct* eigenvalues. The eigenspace S_i contains the eigenvectors associated with λ_i:

$$S_i = \{x \in \mathbb{R} | \mathbf{A}\mathbf{x} = \lambda_i \mathbf{x}\}$$

For real-symmetric matrices, the algebraic multiplicity is equal to the geometric multiplicity, for all the eigenvalues. As of that, the dimension of S_i (which is the geometric multiplicity) is equal to the multiplicity of λ_i. Finally, if $\lambda_i \neq \lambda_j$ then S_i and S_j are mutually orthogonal.

Real-valued functions We consider real-valued functions on the set of the graph's vertices, $f : \mathbb{V} \mapsto \mathbb{R}$. Such a function assigns a real number to each graph node. f is a vector indexed by the graph's vertices, hence $f \in \mathbb{R}^n$. The eigenvectors of the adjacency matrix $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ can be seen as eigenfunctions. The adjacency matrix can also be viewed as an operator

$$\mathbf{g} = \mathbf{A}\mathbf{f} \tag{D.1}$$

$$g(i) = \sum_{i \sim j} f(j) \tag{D.2}$$

or as a quadratic form

$$\mathbf{f}^\top \mathbf{A}\mathbf{f} = \sum_{e_{ij}} f(i) f(j) \tag{D.3}$$

If we fix for each edge in the graph an orientation, we can construct the *incidence matrix* as follows

$$\Delta = \begin{cases} \Delta_{ev} = -1 & \text{if } v \text{ is the tail of } e \\ \Delta_{ev} = 1 & \text{if } v \text{ is the head of } e \\ \Delta_{ev} = 0 & \text{if } v \text{ is not in } e \end{cases}$$

The mapping $\mathbf{f} \mapsto \Delta\mathbf{f}$ is known as the *co-boundary mapping* of the graph.

$$(\Delta\mathbf{f})(e_{ij}) = f(v_j) - f(v_i) \tag{D.4}$$