Artificial Neural Networks: layer architectures, optimizers and automatic differentiation

Filippo Gatti

Université Paris-Saclay
CentraleSupélec, ENS Paris-Saclay, CNRS
LMPS Laboratoire de Mécanique Paris Saclay UMR 9026

This chapter is meant to provide a basic yet solid understanding of artificial neural networks, to a heterogeneous readership. In particular, the chapter presents three major types of neural networks, namely: feed-forward multi layer perceptrons ($\mathcal{MLP}$), convolutional neural networks ($\mathcal{CNN}$) and recurrent neural networks ($\mathcal{RNN}$). The chapter describes how deep is the relationship between each of these neural networks and the specific data science task they aim at tackling: from regression to classification, from images to time-series, with practical tutorials on real datasets and mechanics-inspired examples. The neural network optimization is described in its general statistical framework, focusing on the most popular algorithms tailored to efficiently "train" such neural metamodels, through the so called "back-propagation". The chapter explains how in practice the back-propagation occurs, by automatizing the derivative chain rule and by efficiently exploiting the computational graph constructed to predict. The chapter's last sections provide practical recipes on how to efficiently optimize the learning algorithms, by duly initializing the neural network parameters, by adopting design strategies that avoid vanishing or exploding gradients and by pursuing deeper architectures to achieve data disentanglement, parsimony and generalizability. The chapter is featured by several practical examples, with corresponding code snippets, in order to practice the theoretical aspects presented in the main text. For expert readers, this chapter serves as a recap. We defer to the chapter Artificial Neural Networks: advanced topics, for further more technical and theoretical insights on the vast world of artificial neural networks. The chapter is largely inspired, among others, by Stéphane Mallat's Data Science lecture notes at Collège de France, as well as by different lecture notes of CentraleSupélec's engineering curriculum.

1 Why PyTorch?

The scope of these chapters is to provide a basic understanding of the most important notions of artificial neural networks, for a non-expert readership with a strong background in computational geomechanics. Major theoretical aspects are outlined and supported by detailed examples and tutorials, in order to reinforce the learning curve of basic and advanced concepts with the help of practical and commented hands-on sessions, accessible to everybody. This approach follows the same paradigm followed by the scientific community since a decade ago, when the AI revolution started, thanks to the deployment of cumbersome algorithms on large databases on specific hardware, such as GPUs (Graphic Processing Units) and, from 2016, TPUs (Tensor Processing Units). GPU, followed by TPU, took advantage of powerful CPU (Central Processing Units) because of their proneness to solve the same sequence of operations on a continuous stream of data, progressively offloaded from the host (the CPU, accessing large memory but at a slow pace) to the device (GPU or TPU respectively), with limited memory capacity, but leveraging the possibility of performing massive parallel computations on naively parallel problems (such as processing large databases of independent data realizations). AI and machine learning took advantage of the long tradition in computer vision of leveraging GPU computation to render graphical content. However, in order to exploit such techniques for statistical learning, several libraries have been developed, with alternate fortunes. A decade after the start of the AI revolution in 2012 - the year in which Krizhevsky et al. [KSH17] beat the state-of-art reference in image classification - two major libraries have survived and have become extremely popular for the design of neural networks and complex machine learning techniques: `PyTorch` and `TensorFlow`. The development of those two libraries has been featured by several extra contributions of prototype libraries (`DistBelief`, `Caffe2`, `Theano`, `Keras` to cite a few) that have been progressively integrated in either one of two leading libraries, developed by Meta AI and Google Brain Team respectively.

`PyTorch` was firstly released in 2016, originally developed by Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, as a `python` interface to the `Torch` open-source library, initially created by Idiap Research Institute and École polytechnique fédérale de Lausanne (EPFL) and migrated to `PyTorch` in 2017. `Torch` provides a `LuaJIT` interface to deep learning algorithms written in C. `LuaJIT` leverages the `Lua` lightweight, high-level, multi-paradigm programming language in the context of tracing just-in-time compilation, to optimize the execution of a program at runtime by tracking frequent linear sequences of code, compiling them to native machine code and executing them. On the contrary, `TensorFlow` was released in late 2015 by the Google Brain Team, based on its proprietary ancestor called `DistBelief`, started in 2011. The name `TensorFlow` was inspired by the archetype on which the library is based, i.e. the multidimensional data array referred to as *tensors*. In 2016, Google even developed an application-specific integrated circuit (ASIC), the TPU, specifi-

cally tailored for `TensorFlow` applications and oriented toward using or running models on low floating-point precision (8-bit) rather than training them. Both `PyTorch` and `TensorFlow` are at their second versions, released in 2023 and 2019 respectively.

From a mere technical standpoint, the main differences between the two libraries is the way they assemble computational graphs. Whilst `TensorFlow` distinguishes itself for using static graphs (compiled before running them) to perform back-propagation, `PyTorch` adopts dynamic graph that are assembled at run time. However, in its brand new version 2, recently released, `PyTorch` introduced an experimental tool to compile graphs. The difference between static and dynamic graphs is that the static graphs are based on fixed tensor shapes and graph connections, that allows to optimize the graph operations before running it. However, certain optimization features cannot be applied to dynamic graphs assembled and modified at runtime, possibly reducing their overall speedup. Nevertheless, the `PyTorch`'s dynamic approach eases the coding part for non-expert users, since memory requirements (often unknown) need not to be defined before hand. Static graphs, on the contrary, allow secure model building with high-level API that enables portability on cloud, in the browser, on-device, multiple CPUs and multiple GPUs or TPUs. Static graphs can be used by other supported languages, such as `C++` and `Java`, facilitating cross-platform deployment. On the contrary, `PyTorch` has its own API for saving trained model in a serialized way and its own API for distributed training on parallel architectures (`DistributedDataParallel` or `ModelParallel`.

As described in THE INTRODUCTORY CHAPTER, `TensorFlow` dominated the AI scene at the very beginning (2012-2017) and rapidly left the step to `PyTorch` after release, due to its friendly interface and ease to code, that attired new coming AI practitioners with poor background in machine learning and `python` coding in general. Therefore, the Google team released `TensorFlow` v2.0 in 2019, completely turning v1.0 upside down. The most intriguing novelty was the introduction of the "eager" mode, which introduced the possibility of switching from default static graph computation to the "Define-by-Run". Moreover, `TensorFlow` 2.0 performances on GPU drastically improved, enabling its use to those popular (compared to TPU at least) hardware. This allowed `TensorFlow` to regain a consistent portion of the market.

All in all, both `TensorFlow` and `PyTorch` are incredibly user-friendly and powerful machine learning libraries. `TensorFlow` is definitely more mature, with advanced features for optimizing complex machine learning algorithms. However, `PyTorch` seems more adapt to the ALERT school's purpose of providing a first lookup at the basic AI and neural networks fundamentals.

2 Neural networks

Neural Networks ($\mathcal{NN}$) have shown outstanding capabilities either for classification, either for regression problems. Given a labeled dataset $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$, with $\boldsymbol{x}_k \in \mathcal{X} \subset \mathbb{R}^{d_X}$ and $\boldsymbol{y}_k \in \mathcal{Y} \subset \mathbb{R}^{d_Y}$, the problem they aim at solving is the following:

Problem. ***Empirical loss minimization***$(\mathcal{P})$
Find $\boldsymbol{h_\theta} : \mathcal{X} \to \mathcal{Y}$, $\boldsymbol{h_\theta} \in \mathcal{H_\theta}$ *such that:*

$$\boldsymbol{h}_{\hat{\boldsymbol{\theta}}}\left(\mathcal{D}_{XY}\right) = \underset{\boldsymbol{h_\theta} \in \mathcal{H_\theta}}{\arg\min}\ L_{\mathcal{D}_{XY}}(\boldsymbol{h_\theta})$$

where $\mathcal{H_\theta}$ is the chosen class of classifiers or regression functions (i.e., the neural network architecture), $L_{\mathcal{D}_{XY}}$ the empirical loss (corresponding to the sample average of i.i.d. realizations) defined as:

$$L_{\mathcal{D}_{XY}}\left(\boldsymbol{h_\theta}\right) = \frac{1}{N}\sum_{i=1}^{N} l\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}_k\right), \boldsymbol{y}_k\right), \quad (\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{D}_{XY} \tag{1}$$

with $l\left(\boldsymbol{h}\left(\boldsymbol{x}\right), \boldsymbol{y}\right) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ being a measure of the "distance" between "true" label and prediction, such that $l(\boldsymbol{y}, \boldsymbol{y}) = 0$. $L_{\mathcal{D}_{XY}}\left(\boldsymbol{h}_\theta\right)$ is an approximation of the "true" loss $L\left(\boldsymbol{h_\theta}\right) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p}\left[l\left(\boldsymbol{h}_\theta\left(\boldsymbol{X}\right), \boldsymbol{Y}\right)\right]$ that cannot be computed directly since $p(\boldsymbol{X}, \boldsymbol{Y})$ is not known *a priori*. The theory of information exposed in Section 1.6 draws the analogy between empirical loss $L_{\mathcal{D}_{XY}}$ and the negative log-likelihood of the conditional probability distribution $p_{\boldsymbol{\theta}}\left(y|\boldsymbol{x}\right)$, induced by parametrization of the $\mathcal{MLP}$ $h_{\boldsymbol{\theta}}$ by identifying l with the negative log-likelihood of the conditional probability $p_{\boldsymbol{\theta}}\left(y|\boldsymbol{x}\right)$, defined as follows:

$$l\left(y, h_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right)\right) = -\ln p_{\boldsymbol{\theta}}\left(y|\boldsymbol{x}\right) = \mathcal{NLL}\left(y, h_{\boldsymbol{\theta}}\right) \tag{2}$$

In otherthe negative log-likelihood of . Minimizing $L_{\mathcal{D}_{XY}}$ corresponds therefore to minimize the

A $\mathcal{NN}$ is a particular statistical model $\boldsymbol{h_\theta} \in \mathcal{M_\theta}$ (see Section 1.3) that depends on a set of weights and biases $\theta \in \Theta$. The most common $\mathcal{NN}$ model is the Multi-Layer Perceptron ($\mathcal{MLP}$), that consists into a directed acyclic graph (*feed-forward*), with an input layer, several hidden layers and an output layer of *neurons* (the graph's edges) stacked upon each other (see Figure 1).

Every node in one layer is connected to every other node in the next layer. The $\mathcal{MLP}$ gets deeper as many more hidden layers are stacked-up. The fundamental elementary brick of any $\mathcal{MLP}$ is the *neuron*.

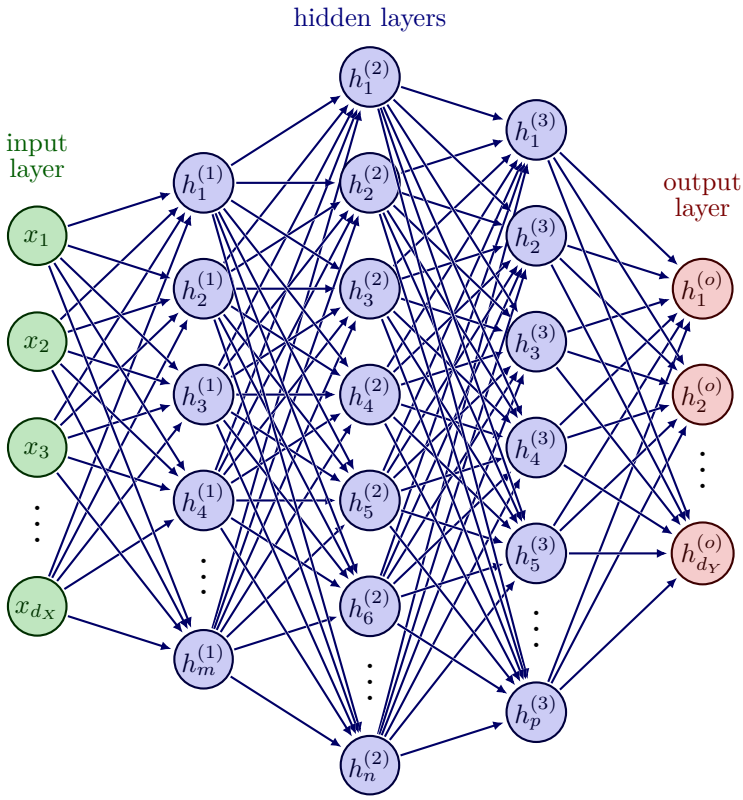


Figure 1: Simple example of Multi-Layer Perceptron (3 hidden layers $\boldsymbol{h}^{(\ell)}$ and 1 output $\boldsymbol{y}$). The figure was drawn with tikz.net.

2.1 The artificial neuron

Artificial neural networks have been formulated by the pioneering work of Pitts and McCulloch [MP43] and Rosenblatt [ROS57]. A *neuron* (or unit) essentially performs the following non-linear transformation on its input $\boldsymbol{x}$:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = g(a(\boldsymbol{x})) = g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b) = g\left(\sum_{c=1}^{d_X} w_c \cdot x_c + b\right); \qquad \theta := \{\boldsymbol{w}; b\} \quad (3)$$

where a is the linear *pre-activation*, $\boldsymbol{w}$ the weights and b the bias. g is the non-linear *activation function* and it can be chosen in the list (non-exhaustive):

- Sigmoid:

$$g(a) = \sigma(a) = \frac{1}{1 + e^{-a}} \quad (4)$$

Properties of the Sigmoid function:

- it is bounded in $[0; 1]$
- it is positive
- it is strictly increasing
- it serves as probability distribution, i.e., $g(a) = P_Y[Y = a|\boldsymbol{x}]$
- it has a "squashing" effect
- it is a smooth version of binary classifier with linear decision boundary (Logistic regression)

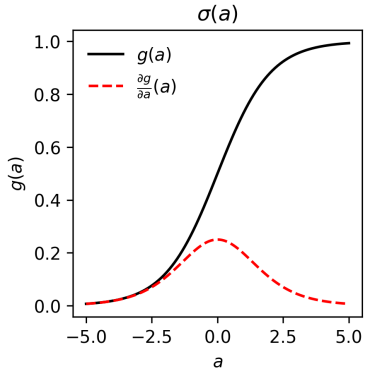


Figure 2: Sigmoid activation function $g(a)$ and its derivative $\frac{\partial g}{\partial a}$.

- Hyperbolic Tangent:

$$g(a) = \tanh(a) = \frac{e^{2a} - 1}{e^{2a} + 1} \tag{5}$$

Properties of the Hyperbolic Tangent:

- it is bounded in $[-1, 1]$
- it is skew-symmetric
- it is strictly increasing
- it has a "squashing" effect (vanishing gradients, see Section 4.2)
- it is preferred to Sigmoid because it has 0 steady state ($\tanh(0) = 0$) [LeC+98; GB10].

- Rectified Linear Unit [GBB11]:

$$g(a) = ReLU(a) = \max(0, a) \tag{6}$$

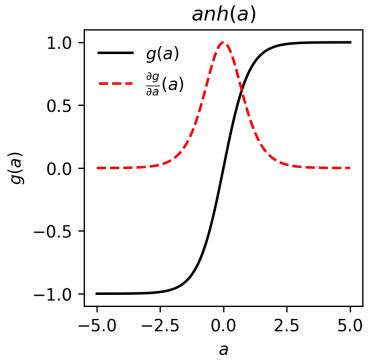


Figure 3: Hyperbolic tangent activation function $g(a)$ and its derivative $\frac{\partial g}{\partial a}$.

Properties of the Rectified Linear Unit:

- closer to real biological activity since indicates that cortical neurons are rarely in their maximum saturation regime [GBB11]
- it is bounded to 0 by below for $a < 0$
- it is positive $\forall a \in \mathbb{R}$
- it is strictly increasing $\forall a \in \mathbb{R}$
- it does not enforce a sign symmetry or antisymmetry
- it promotes sparsity in combination with a ℓ^1-norm penalty on the weights (see Section 2.3 and [GBB11])
- the deactivation for $a < 0$ is robust to noise

- Leaky Rectified Linear Unit:

$$g(a) = LeakyReLU(a) = \begin{cases} a, a \geq 0 & (7) \\ -\alpha \cdot a, a < 0 (\alpha > 0) & (8) \end{cases}$$

Properties of Leaky Rectified Linear Unit:

- it is not bounded
- it maintains the sign of a
- it has no "squashing" effect

- Exponential Linear Unit [CUH16]

$$g(a) = ELU(a) = \begin{cases} a, a > 0 & (9) \\ \alpha \cdot (e^a - 1), a \leq 0 & (10) \end{cases}$$

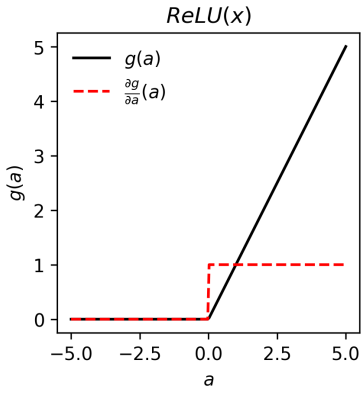


Figure 4: $ReLU$ activation function $g(a)$ and its derivative $\frac{\partial g}{\partial a}$.

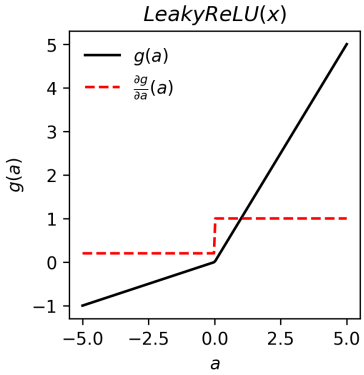


Figure 5: $LeakyReLU$ activation function $g(a)$ and its derivative $\frac{\partial g}{\partial a}$.

Properties of the Exponential Linear Unit:

- it equals the identity for $a > 0$
- it is not bounded for $a > 0$
- it is strictly increasing $\forall a \in \mathbb{R}$
- it has a "squashing" effect for $a \leq 0$
- noise-robust deactivation state ($\boldsymbol{\mu}_w \to \mathbf{0}$) [CUH16] (see Section 4.2)

- Scaled Exponential Linear Unit [Kla+17]

$$g(a) = SELU(a) = s\left(\max(0, a) + \min\left(0, \alpha\left(e^a - 1\right)\right)\right) \tag{11}$$

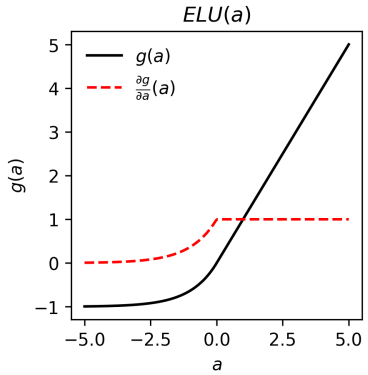


Figure 6: ELU activation function $g(a)$ and its derivative $\frac{\partial g}{\partial a}$.

Properties of the Scaled Exponential Linear Unit:

- α =1.67326324235437728481704299167 17
- s =1.05070098735548049341933498529 46
- identity for $a > 0$
- bounded to $-\alpha$ by below for $a < 0$
- self-normalizing weights ($\boldsymbol{\mu}_w \to \mathbf{0}$, $\mathbb{C}_{\boldsymbol{w},\boldsymbol{w}} \to \mathbf{I}$) even for noisy input [Kla+17] (see Section 4.2)

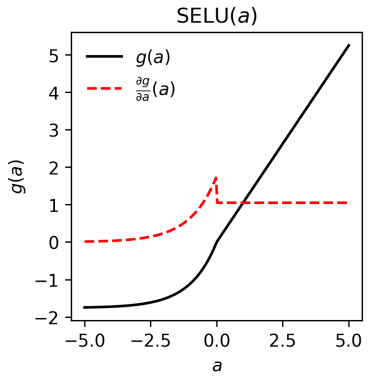


Figure 7: $SELU$ activation function $g(a)$ and its derivative $\frac{\partial g}{\partial a}$.

The functions $g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle)$ are *ridge functions*, which is constant on the hyperplane $a(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle = c$ and on each hyperplane $a(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = c$ (see

Figure 8).

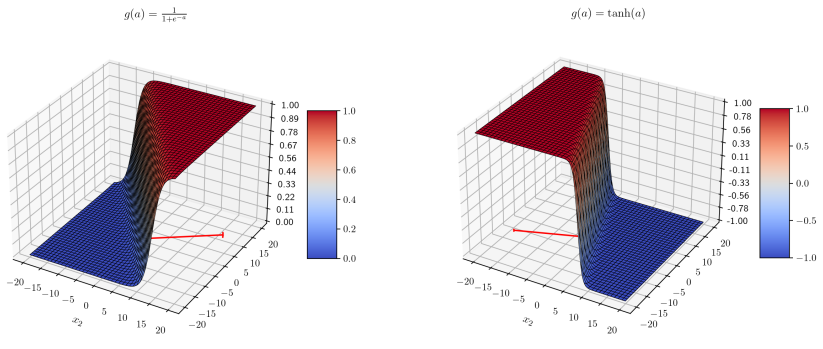


Figure 8: Examples of linear separation made by a single neuron on an input dataset $\mathcal{X} \subset \mathbb{R}^2$. (a) activation function $g(a) = \sigma(a)$; (b) activation function $g(a) = \tanh(a)$.

From Figure 8, one notices that b makes the ridge $h_{\boldsymbol{\theta}}$ translate along the input space, whereas $\boldsymbol{w}$ determines the orientation of the ridge (in-plane rotation). The activation function adds non-linearity and if $g(\cdot) = \sigma(\cdot)$, $h_{\boldsymbol{\theta}}$ can be interpreted as the likelihood $p_{\boldsymbol{\theta}}(\boldsymbol{x})$. In a binary classification context with $y \in \mathcal{Y} := \{0, 1\}$ (if $g(\cdot) = \sigma(\cdot)$) or $y \in \mathcal{Y} := \{-1, 1\}$ (if $g(\cdot) = \tanh(\cdot)$), the combination of b, $\boldsymbol{w}$ and g determines *decision boundary*, i.e., the hyperplane that separates the samples of class $y = 0$ (or $y = -1$) if $h_{\boldsymbol{\theta}}(\boldsymbol{x}) > 0.5$ from those labeled as $y = 1$, if $h_{\boldsymbol{\theta}}(\boldsymbol{x}) \leq 0.5$. In a regression context, the combination of b, $\boldsymbol{w}$ and g represents a non-linear ridge regression.

Example 1. **Create a single neuron network in `PyTorch`**

```
import torch
import torch.nn as nn # library for predefined NN classes

class neuron(nn.Module):
    def __init__(self):
        super(neuron,self).__init__(x_size=28*28, h_channels=10)
        # hidden nodes in each layer
        self.x_size = x_size
        self.h_channels = h_channels

        # This is the 1st hidden layer (784 -> hidden_1)
        self.a = nn.Linear(self.x_size, self.h_channels)
        self.g = nn.ReLU(inplace=False)
        # Inplace in the code explains how the function
        # should treat the input. Inplace as true replaces
        # the input to output in the memory. Though this helps
        #  in memory usage, this creates problems for the code
        # being used as the input is always getting replaced as
        # output. It is better to set in place to false as this
        # helps to store input and output as separate storage
```

```
        # spaces in the memory.

        # The dropout layer (p=0.2)
        self.droput = nn.Dropout(0.2)

    def forward(self,x):
        # flattening the input image
        x = x.view(-1,self.x_size)

        # adding the hidden layer, for activation we are using relu activation
        ho = self.droput(self.g1(self.a1(x)))

        return ho

x_size = 28*28
y_size = 10
h_theta = neuron(x_size, y_size)
```

2.2 The Multi-Layer Perceptron

A single neuron is not sufficient when the decision boundary is not linear. For instance, in a classification problem with input $\boldsymbol{x} = (x_1, x_2) \in \{0,1\}^2$ and associated true label $y = f(\boldsymbol{x})$ a neuron such as the one described in Figure 8 can successfully draw a linear decision boundary and separate the dataset into two clusters if $f = f_{OR}(\boldsymbol{x}) = \{x_1\} \cup \{x_2\}$, $f = f_{AND1}(\boldsymbol{x}) = \{x_1\} \cap \{\{0,1\} / \{x_2\}\}$ and $f = f_{AND2}(\boldsymbol{x}) = \{\{0,1\} / (x_1)\} \cap \{x_2\}$ (see Figures 9b to 9d). However, one neuron cannot linearly separate the data labeled with the function $f = f_{XOR}(\boldsymbol{x}) = \{\{x_1\} \cup \{x_2\}\} / \{\{x_1\} \cap \{x_2\}\}$ as shown in Figure 9d.

Instead of one single neuron, the classifier $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ can be improved by stacking two layers of neurons one upon each other: a *hidden layer* made of two neurons and an *output layer*, made of one neuron only (the architecture is depicted in Figure 10a). As shown in Figure 10b, $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ can now successfully separate the boundary induced by f_{XOR}.

But how is that possible? Adding hidden layers reveals to be crucial to raise the complexity of the predictor, thanks to intermediate transformations $\boldsymbol{\phi} : \boldsymbol{x} \mapsto \mathbb{R}^{d_\phi}$ that map the input $\boldsymbol{x}$ into the space of *hidden features* of size d_ϕ:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = g(a(\boldsymbol{\phi}(\boldsymbol{x}))) = g(\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}) \rangle + b) = g\left(\sum_{c=1}^{d_\phi} w_c \cdot \phi_c(\boldsymbol{x}) + b \right); \qquad \theta := \{\boldsymbol{w}; b\} \tag{12}$$

In the case of XOR classification, the architecture in Figure 10a is conceived in order to map $\boldsymbol{x}$ into the intermediate hidden representations, one per hidden neuron, $h_1(\boldsymbol{x}) = f_{AND1}(\boldsymbol{x})$ and $h_2(\boldsymbol{x}) = f_{AND2}(\boldsymbol{x})$. Figure 10b show that the XOR decision boundary is linear in the (h_1, h_2) plane.
It is clear that the choice of $\boldsymbol{\phi}$ adds flexibility to the predictor. The size and properties of this intermediate representations can be chosen via regression *kernels* $\Bbbk : \mathcal{X} \times \mathcal{X} \mapsto \mathcal{X}$ such that $\int_{\mathbb{R}^{d_\mathcal{X}}} \Bbbk(\boldsymbol{x}_1, \boldsymbol{x}_2) \boldsymbol{\phi}(\boldsymbol{x}_2) d\boldsymbol{x}_2 = \lambda \mathbb{M}^{-1} \boldsymbol{\phi}(\boldsymbol{x}_1)$

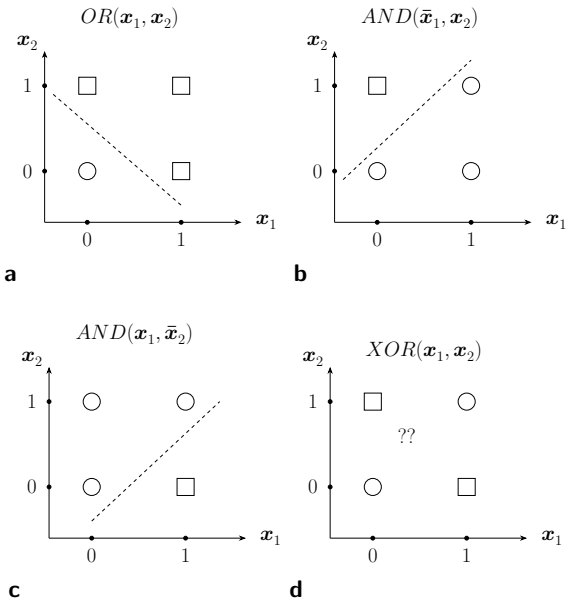


Figure 9: Solved and unsolved decision boundaries for different binary operations. (a) $f_{OR}(\boldsymbol{x}) = \{x_1\} \cup \{x_2\}$, (b) $f_{AND1}(\boldsymbol{x}) = \{x_1\} \cap \{\{0,1\} / \{x_2\}\}$, (c) $f_{AND2}(\boldsymbol{x}) = \{\{0,1\} / (x_1)\} \cap \{x_2\}$, (d) $f_{XOR}(\boldsymbol{x}) = \{\{x_1\} \cup \{x_2\}\} / \{\{x_1\} \cap \{x_2\}\}$ (unsolved). Reprinted from the video-lecture notes by Hugo Larochelle (https://www.youtube.com/watch?v=iT5P4z6Fzj8&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=3).

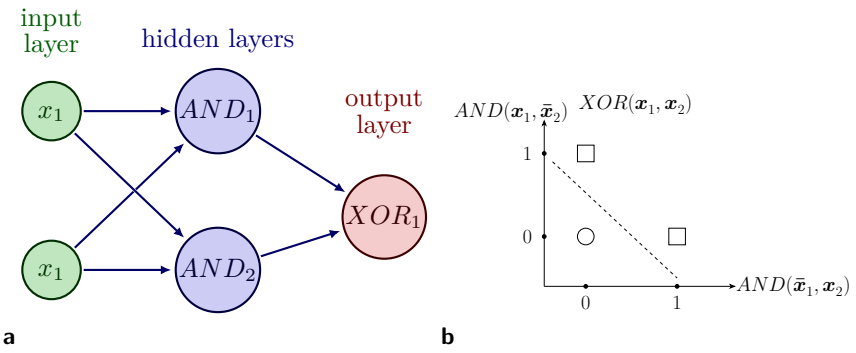


Figure 10: Simple example of Multi-Layer Perceptron (1 hidden layers $\boldsymbol{h}^{(1)}$ and 1 output $y \in \mathbb{R}$). layer. The figure was drawn with tikz.net.

with a metric $\mathbb{M}$ such that $\int_{\mathbb{R}^{d_X}} \langle \boldsymbol{\phi}(\boldsymbol{x}), \mathbb{M}^{-1} \boldsymbol{\phi}(\boldsymbol{x}) \rangle \, d\boldsymbol{x} = 1$ (see Chapter 2 - *Introduction to regression methods*).

Alternatively, one can stack several layers of neurons one upon each other, so that the final predictor results in a composition of hidden layers $\boldsymbol{h}^{(\ell)}$ and one output function $\boldsymbol{h}^{(o)}$. For instance, in case of one hidden neuron, the $\mathcal{NN}$ output reads:

$$
\begin{aligned}
h_{\boldsymbol{\theta}}(\boldsymbol{x}) = h^{(o)} \circ \boldsymbol{h}^{(1)}(\boldsymbol{x}) = g^{(o)}(a^{(o)}(\boldsymbol{h}^{(1)}(\boldsymbol{x}))) = g\left(\left\langle \boldsymbol{w}^{(o)}, \boldsymbol{h}^{(1)}(\boldsymbol{x}) \right\rangle + b^{(o)}\right) = \\
= g^{(o)}\left(\sum_{c=1}^{d_{h_1}} w_c^{(o)} \cdot h_c^{(1)}(\boldsymbol{x}) + b^{(o)}\right); \quad \theta := \left\{\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(o)}; b^{(1)}, b^{(o)}\right\}
\end{aligned}
\tag{13}
$$

with

$$
\begin{aligned}
h_c^{(1)} = g^{(1)}(a_c^{(1)}(\boldsymbol{x})) = g^{(1)}\left(\left\langle \boldsymbol{w}^{(1)}, \boldsymbol{x} \right\rangle + b^{(1)}\right) \\
= g^{(1)}\left(\sum_{c=1}^{d_X} w_c^{(1)} \cdot x_c + b^{(1)}\right)
\end{aligned}
\tag{14}
$$

More in general, $\mathcal{MLP}$ gathers several layers of neurons, with each neuron c in a layer ℓ being fully-connected (a connection means a composition) with all the neurons in the layer $\ell-1$ - as its input - and with all the neurons in layer $\ell+1$, as its output. Figure 11 shows an example of fully-connected $\mathcal{MLP}$, focusing on the connections between a hidden layer ℓ and the successive one layer $\ell+1$. For each connection c, a weight vector $\boldsymbol{w}_c$ and a bias b_c are adopted. The weight vectors, stacked column-wise, constitute the weight matrix $\boldsymbol{W}$, with $\boldsymbol{W}_{c,:} = \boldsymbol{w}_c$.

Some crucial aspects of a fully-connected (or dense) feed-forward $\mathcal{NN}$ such as the standard $\mathcal{MLP}$ must be noted:

- the activation function $g^{(\ell)}$ is the same for all neurons belonging to the same layer ℓ;
- all neurons (or units) belonging to the ℓ^{th} layer receive the same input, i.e., a the vector $\boldsymbol{h}^{(\ell-1)}$ of size $u^{(\ell)}$[1], representing the input units to the ℓ^{th} layer of the graph;
- the output of each of the $u^{(\ell+1)}$ neurons the ℓ^{th} layer neuron (or unit) belonging to the ℓ^{th} layer $h_i^{(k)}$ of size [2], representing the input units of the graph.

Example 2. **Create a single neuron network in `PyTorch`**

[1] $u^{(\ell)}$ is commonly indicated in the literature as fan_{in}
[2] $u^{(k)}$ is commonly indicated in the literature as fan_{in}

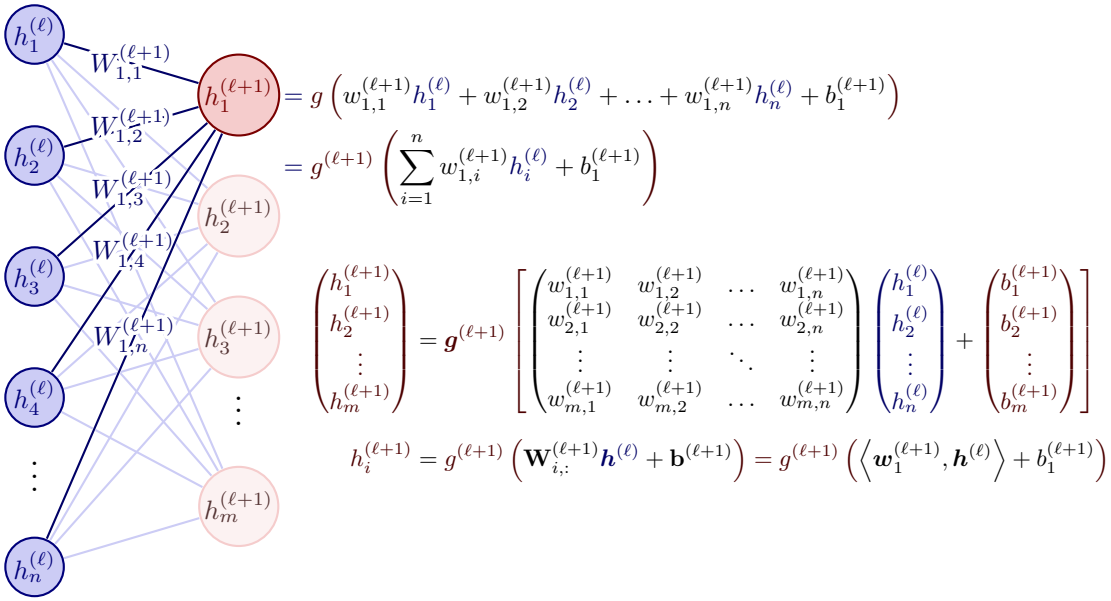


Figure 11: Scheme of the last output layer of a fully-connected Multi Layer Perceptron. The figure was drawn with tikz.net.

```python
import torch
import torch.nn as nn # library for predefined NN classes

class MLP(nn.Module):
  def __init__(self):
    super(MLP,self).__init__(x_size=28*28, y_size=10)
      # hidden nodes in each layer
      self.x_size = x_size
      self.y_size = y_size
      self.hn_1 = 512
      self.hn_2 = 512

      # This is the 1st hidden layer (784 -> hidden_1)
      self.a1 = nn.Linear(self.x_size, self.hn_1)
      self.g1 = nn.ReLU(inplace=False)
      # Inplace in the code explains how the function
      # should treat the input. Inplace as true replaces
      # the input to output in the memory. Though this helps
      #  in memory usage, this creates problems for the code
      # being used as the input is always getting replaced as
      # output. It is better to set in place to false as this
      # helps to store input and output as separate storage
      # spaces in the memory.

      ## This is also linear layer but (n_hidden -> hidden_2)
      self.a2 = nn.Linear(self.hn_1, self.hn_2)
      self.g2 = nn.ReLU(inplace=False)

      # This is the linear output layer with (n_hidden -> 10)
      self.ao = nn.Linear(self.h2_n, self.y_size)

```

```
        # The dropout layer (p=0.2)
        self.droput = nn.Dropout(0.2)

    def forward(self,x):
        # flattening the input image
        x = x.view(-1,self.x_size)

        # adding the hidden layer, for activation we are using relu activation
        h1 = self.droput(self.g1(self.a1(x)))

        # adding the hidden layer, for activation we are using relu activation
        h2 = self.droput(self.g1(self.a1(h1)))

        # adding the output layer
        ho = self.a3(h2)

        return ho

x_size = 28*28
y_size = 10
h_theta = MLP(x_size, y_size)
```

2.3 Possible solutions to the curse of dimensionality

According to [Mai99] (see Theorem 27), the number of hidden neurons N_K demanded to assure that a $\mathcal{MLP}$ with a sole hidden layer approximates a function $f : \boldsymbol{x} \mapsto y$, provided that at least $f \in W^{k,2}$[3] and with an accuracy ε is of the order of:

$$N_K \approx \varepsilon^{\frac{1-d_X}{k}} \tag{15}$$

with $d_X = \dim(\mathcal{X})$, and k being the highest order of weak derivative of f. The *curse of dimensionality* of the Universal Approximation Theorem (expressed in Theorem 24) resides in the fact that the larger is the dimension of the data space $\mathcal{X}$ and the poorer is the regularity of the function (i.e., the lower is the highest order of weak derivative k), the larger would be the number of hidden neurons of a 1-hidden-layer $\mathcal{MLP}$ demanded to approximate the labeling or regression function f. [Bac17] proposed an interesting strategy to break the curse of dimensionality presented in Section 2.1. The strategy consists in several countermeasures that are detailed in the following paragraphs

Reduce the dimensionality In other words, one should select a subset $\mathcal{S} \subset \mathcal{X}$ with dimension $d_S < d_X$. In this sense, one possible strategy is the Principal Component Analysis (PCA) or other Reduced Order Methods (ROM). PCA and other linear methods for dimensionality reduction are adopt an linear transform of the type $\boldsymbol{a}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$, with $\boldsymbol{W} \in \mathcal{M}_{(d_S,d_X)}(\mathbb{R})$, with $\text{rank}(\boldsymbol{W}) = d_S < d_X$ (see related chapters for further detail).

Separate the interactions (disentanglement) The separation of interaction is a widely adopted assumption in physics. It postulates that the function

[3] $W^{k,2}$ is the Sobolev space defined in Equation (234)

$f : \mathcal{X} \to \mathbb{R}$ is a the sum of functions f_i acting on local subdomains. For instance, the Markov processes assume the separability of the interaction, since they postulate that the transition probability from a state to another strictly depend on those two states only. Future and long-time past interactions are neglected. The separability of interaction can be formulated as follows:

$$\exists \{f_n\}_{n\in I}, f_n : \mathcal{X}_n \to \mathbb{R}, \mathcal{X}_n \subset \mathcal{X} \quad f(\boldsymbol{x}) = \sum_{n\in I} f_n(\boldsymbol{x})|_{\boldsymbol{x}\in\mathcal{X}_n}, \forall \boldsymbol{x} \in \mathcal{X} \tag{16}$$

The separability of interactions is also called *disentanglement*. Disentangling the data representations consists into mapping the data set into a latent representation (see Section 1.6) spanned by a basis that depends on the so called *factor of variation* of the data set at stake [KM18]. For instance, a dataset composed of force-displacements values issued from a traction test on a steel sample of Young modulus E, under different temperature values T (within the limits of thermoelasticity), could ideally be represented by a latent representation $\boldsymbol{z} = z_E \boldsymbol{e}_E + z_T \boldsymbol{e}_T$, where z_E only varies whenever the Young's modulus of the sample changes, at fix temperature, whereas z_T changes whenever the same sample undergoes the traction test at different temperature. More in general, assuming that the database can be generated by a set of *semantic meaningful features*, like Young's modulus and temperature in the previous examples, the disentanglement process aims at separating the effect of each factor - in the latent representation - in order to structure the latent space in an interpretable way [Fra+22]. In machine learning, datasets consist of multiple i.i.d. samples, assumption that allows to assume a factorized parametric probability distribution $p_{\boldsymbol{\theta}}(x_1, \ldots, x_N) = \prod_{i=1}^N$ and therefore to maximize the log-likelihood of the observations (see Section 1.3). The idea of disentanglement is to achieve a latent representation of the dataset (to "encode" the data), associated to a parametric probability distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, that extract the common features of the dataset and represents them into a disentangled way, e.g., by forcing a factorized probability distribution on $\boldsymbol{z} \sim \prod_{i=1}^N q_{\boldsymbol{\phi}}(z_i|\boldsymbol{x})$. However, this strategy is not sufficient to achieve the disentanglement of the latent representation, since all z_i could vary when changing one single factor of variation in the data set [Fra+22].

Adopting the reduction of dimensionality, one can intuitively look for subdomains $\mathcal{X}_n$ of dimension $d_S < d_X$. In this case, the approximation can be done by subdomains, employing the same 1-hidden-layer $\mathcal{MLP}$ with a total amount of neurons $N_K \approx \varepsilon_n^{\frac{1-d_S}{k}}$ (see Equation (15)). The total error would be estimated as

$$\varepsilon = \sum_{n\in I} \varepsilon_n \approx \mathrm{card}(I) \cdot N_K^{\frac{k}{1-d_S}} \qquad N_K \approx \left(\frac{\varepsilon}{\mathrm{card}(I)}\right)^{\frac{1-d_S}{k}} \tag{17}$$

The most intuitive way of separate the interaction is to decompose the function

along each component of the data space, i.e.:

$$\exists \{f_n\}_{n=1}^{d_X}, f_n : \mathbb{R} \to \mathbb{R} \quad f(\boldsymbol{x}) = \sum_{n=1}^{d_X} f_n(w_n \cdot x_n), \forall \boldsymbol{x} \in \mathcal{X}$$

With this strategy, d_S =rank$(w_n) = 1$.

Promote sparsity

> *[...] pour ranger les êtres sous des dénominations communes, et générique, il en falloit connoître les proprietés et les différences; il falloit des observations, et des définitions [...]*
> J.J. Rousseau, 1772 [Rou72]

Rousseau, in his "*Discours sur l'origine et les fondemens de l'inégalité parmi les hommes*", published in 1772, suggested the need of a *dictionary* to describe the reality. The hidden features that describe the function regularity can be effectively embedded into a dictionary $D = \{(k, v(\boldsymbol{x}))_n\}_{n \leq N_D}$ (k is the key and v the corresponding value), with:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{n \in I} \theta_n [k_n] v_n(\boldsymbol{x}), \quad \text{card}(\mathrm{I}) \leq N_D \tag{18}$$

Rousseau intuited that, the size of the dictionary N_D decreases with the knowledge of the phenomenon:

> *[...] plus les connaissances étoient bornées, et plus le Dictionnaire devint étendu.*
> J.J. Rousseau, 1772 [Rou72]

As a matter of fact, the dictionary embedding requires the error to decay rapidly with the cardinality of I $\subseteq$ D, keeping this cardinality small. In other words, we seek for an error decaying as and $\varepsilon = \text{card}(\mathrm{I})^{-\beta}$. This approach is not related to the notion of regularity of the function as intedend in the Fourier approach, i.e., as the largest order of derivation with continuous derivative. Moreover, the fact that the coefficients $\theta[k_n]$ depend on the key value k_n, the approximation in Equation (18) is non-linear and adaptive: non-linear since the dictionary depends on f, adaptive because the choice of I is abitrary [Cam19]. Moreover, compared to the infinite number of weights $\boldsymbol{w}_n \in \mathbb{Z}^{d_X}$ necessary to approximate f with its Fourier series, with the dictionary embedding approach we hope to promote parsimony by considering card(I) low. In particular, [Bar93] proposes to focus on functions with L^1 first order derivative (see Theorem 28). Penalizing the learning algorithm with a ℓ^1 norm on the weights $\boldsymbol{\theta}$ allows to promote sparsity and achieve parsimony. In other words, the 1-hidden-layer $\mathcal{MLP}$ will learn how to approximate the labeling function with the least amount of hidden features and with an approximation error that is not dependent on the dimension of the input space (or alternatively, for a fixed accuracy) the design

of the $\mathcal{NN}$ will require an amount of hidden neurons that is independent of the dimension of the data space.
Among the other advantages of sparsity, it must be mentioned the fact that the data variation is better disentangled and therefore it helps separate the interactions (as outlined in Section 2.3) and to adopt the strictly necessary number of keywords for the dictionary in Equation (18). Nevertheless, forcing too much sparsity may limit the predictive performance of the $\mathcal{NN}$ for an equal number of neurons, because it reduces the effective capacity of the model [GBB11].

2.4 How to improve the $\mathcal{MLP}$ accuracy?

Despite the fact that Theorem 24 proves the universal approximation capability of a 1-hidden-layer $\mathcal{MLP}$, provided that enough neurons are considered, this result is quite hard to exploit in real applications, since the number of neurons can easily become too large to handle fro a computational standpoint. The natural question that arises is: what kind of functions cannot be approximated with an arbitrary accuracy by a $\mathcal{MLP}$ of N_ℓ layers? Moreover, how many neurons should be considered for each layer? What is the effect of having a number of layers that is higher than the number of neurons per layer? The answers to these questions were provided by Eldan and Shamir [ES16] showed that it exists an approximately radial function $\varphi(\|\boldsymbol{x}\|) : \mathbb{R}^{d_X} \to \mathbb{R}$ that can be approximated by a "small" (bounded number of neurons) 2-hidden-layers $\mathcal{MLP}$ with arbitrary accuracy, but that cannot be approximated by a 1-hidden-layer $\mathcal{MLP}$ below a certain accuracy, unless the number of neurons N_K grows exponentially with d_X. In particular, this results is valid for any activation function g and with no further constraint on the weights and biases adopted in the $\mathcal{MLP}$ (on the contrary, the Universal Approximation Theorem requires that the high-frequency components $\|\boldsymbol{w}_n\|$ are smaller than a constant). This results proves that increasing the depth of the $\mathcal{MLP}$ widens the approximation capability of the $\mathcal{MLP}$ and that the depth of the $\mathcal{MLP}$ should be privileged with the respect to its layer dimension (but being careful to avoid vanishing gradient problems, as discussed in Section 4.2. The approximately radial function is the inverse Fourier transform of the indicator function on a unit volume euclidean ball $\mathrm{B}_{d_X}(\boldsymbol{w})$, of radius R_{d_X} such that $R_{d_X} \cdot \mathrm{B}_{d_X}$ has unit volume and it reads:

$$\varphi(\boldsymbol{x}) = \left(\frac{R_{d_X}}{\|\boldsymbol{x}\|}\right)^{\frac{d_X}{2}} J_{\frac{d_X}{2}}(2\pi R_{d_X}\|\boldsymbol{x}\|) = \int_{\boldsymbol{w}:\|\boldsymbol{w}\|\leq R_d} e^{-2\pi i\langle \boldsymbol{x},\boldsymbol{w}\rangle} d\boldsymbol{w} \tag{19}$$

with $J_{\frac{d_X}{2}}(2\pi R_{d_X}\|\boldsymbol{x}\|)$ the Bessel function of first kind of order $\frac{d_X}{2}$. $\varphi(\boldsymbol{x})$ has a high-frequency symmetry by revolution, as depicted in Figure 12. Since the function peaks in $\boldsymbol{x} = \boldsymbol{0}$, the choice of training samples $(\boldsymbol{x}_i, \varphi(\boldsymbol{x}_i))$ must be done carefully. In Figure 13, 4000 samples uniformly distributed in $[-10, 10]^2$ were chosen, 70% of which used for training and 30% for testing purposes. According to [ES16], the number of high-frequency components $\boldsymbol{w}_n$ necessary

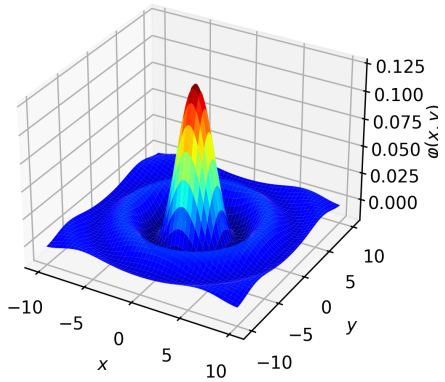


Figure 12: Function $\varphi(\boldsymbol{x}) = \left(\frac{R_{d_X}}{\|\boldsymbol{x}\|}\right)^{\frac{d_X}{2}} J_{\frac{d_X}{2}}(2\pi R_{d_X}\|\boldsymbol{x}\|)$, defined in [ES16], with $d_X = 2$ and $R_d = 0.2$.

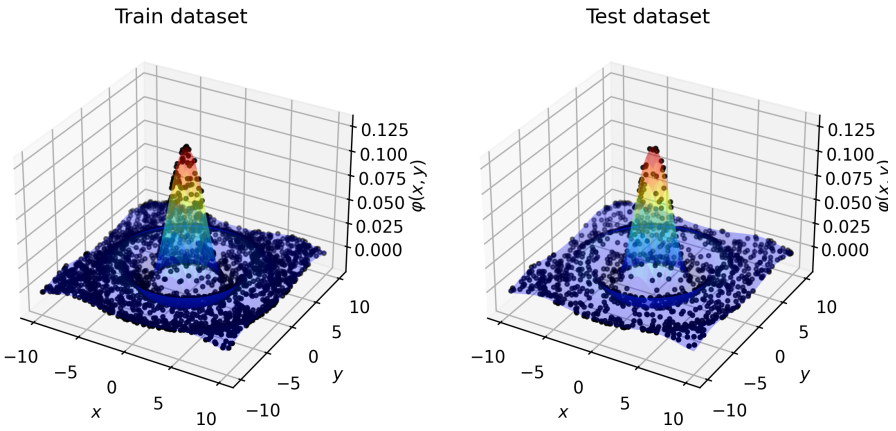


Figure 13: 4000 train and test samples $(\boldsymbol{x}_i, \varphi(\boldsymbol{x}_i))$ to train a $\mathcal{MLP}$ that approximates the function $\varphi(\boldsymbol{x}_i)$ defined in Equation (19). Each instance was uniformly sampled over $[-10, 10]^2$.

to approximate such a function with a one-hidden-layer $\mathcal{MLP}$ (referred as to $h^1_{\boldsymbol{\theta}}$) is very large, which prevents the choice of a parsimonious representations by thresholding the Fourier coefficients such that $\|\boldsymbol{w}_n\| > C$. On the contrary, the revolution symmetry is captured by adding a second hidden layer to the standard 1-hidden-layer $\mathcal{MLP}$, obtaining a 2-hidden-layers $\mathcal{MLP}$ referred as

to $h^2_{\boldsymbol{\theta}}$. In order to numerically prove this statement, Figure 14 shows the L^2 error $\|\varphi(\boldsymbol{x}) - h_{\boldsymbol{\theta}}(\boldsymbol{x})\|^2$ evolution, along training epochs (for first order gradient descent algorithm with `AdamW` optimizer, see Section 3.4.5) for both $h^1_{\boldsymbol{\theta}}$ (featured by 10000 neurons in its single hidden layer, with $ReLU$ activation functions) and $h^2_{\boldsymbol{\theta}}$ (featured by 100 neurons for each of the 2-hidden layers, with $ReLU$ activation functions).

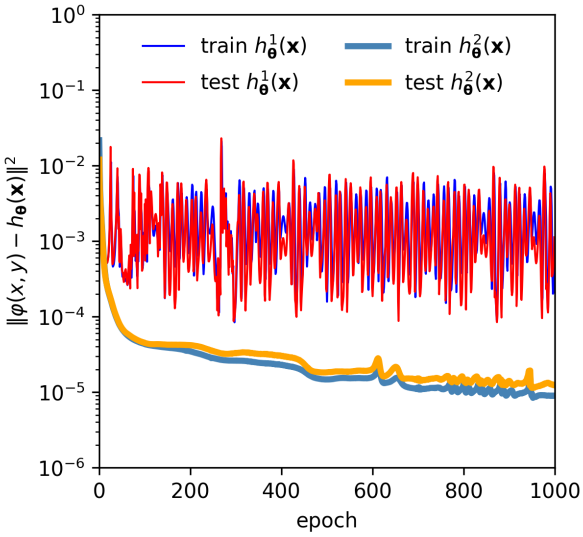


Figure 14: Learning curves for $h^1_{\boldsymbol{\theta}}$ (1-hidden-layer $\mathcal{MLP}$) and $h^2_{\boldsymbol{\theta}}$ (2-hidden-layers $\mathcal{MLP}$), representing the evolution of $\|\varphi(\boldsymbol{x}) - h_{\boldsymbol{\theta}}(\boldsymbol{x})\|^2$ with training epoch. Blue and red curves represent the error evolution $\|\varphi(\boldsymbol{x}) - h^1_{\boldsymbol{\theta}}(\boldsymbol{x})\|^2$, for train and test datasets respectively. Light blue and orange thick curves represent the error evolution $\|\varphi(\boldsymbol{x}) - h^2_{\boldsymbol{\theta}}(\boldsymbol{x})\|^2$ instead, for train and test datasets respectively.

Figure 14 shows the net improvement (one two orders of magnitude) obtained by adding one extra hidden layer to the standard 1-hidden-layer $\mathcal{MLP}$: not only the error decreases for both train and test dataset, but the number of neuron required per layer is 100 times lower for $h^2_{\boldsymbol{\theta}}$. Despite the large number of neurons, the $h^1_{\boldsymbol{\theta}}$ $\mathcal{MLP}$ cannot efficiently minimize the mean-square error, proving the theoretical findings by Eldan and Shamir [ES16].

2.5 From neurons to filters

Given its universal approximation capability (see Theorem 24), the use of $\mathcal{MLP}$ is rather attractive. However, in order to pursue a general criterion of parsimony, some specific *symmetries* can be exploited. For instance: classifying the image should not depend on any affine transformation (i.e., roto-translation). Exploiting symmetries (or more in general *group invariance*) is a valid strategy to face the curse of dimensionality. The statistical model must avoid overparametrization (that leads to overfitting) due to non invariant representations of the dataset with the respect to those symmetries [Cam20].
To better understand this aspect, let us consider a collection of labeled images $\mathcal{D}_{XY} = \{\mathbb{X}_k, y_k\}_{k=1}^N$ of $W \times H$ pixels each. Any image $\mathbb{X}_k$ of the database is composed by a set of pixel and each pixel is associated to a color, expressed by a 3-dimensional vector on the Red Green Blue (RGB) scale. Therefore, $\mathbb{X}_k \in \mathbb{R}^{W \times H \times 3}$. For the sake of simplicity, the image is spanned over a $[0,1]^2$ domain, by a regular grid of $W \times H$ pixels with a 2D index $\boldsymbol{x} = x_1\boldsymbol{e}_1 + x_2\boldsymbol{e}_2, (x_1, x_2) \in [0,1]^2$ and with the color being represented by $\mathbb{X}(x_1, x_2, :) = \boldsymbol{\psi}(x_1, x_2) = \psi_R(x_1, x_2)\,\boldsymbol{e}_1 + \psi_G(x_1, x_2)\,\boldsymbol{e}_2 + \psi_B(x_1, x_2)\,\boldsymbol{e}_3$ (see the code snippet below and Figure 15).

Example 3. **Convert an image to `PyTorch` tensor**
The example that follows parses and converts a digital image $\mathbb{X}$ of the geological cross-cut underneath the Kashiwazaki-Kariwa nuclear power plant (see [CGL21]).

```
import torch
import torchvision
from torchvision.transforms import ToTensor
from PIL import Image

# parse image
with Image.open('./geology.tiff') as image:
    # convert image to RGB
    image=image.convert("RGB")

# convert image to tensor
X = ToTensor()(image).unsqueeze(0)
print(X.size)
# >>> torch.Size([1, 3, 188, 257])
```

y_k belongs to an alphabet A for a classification purpose and it is usually coded as a *one-hot vector*, i.e. a binary vector of the size of the alphabet. The framework can be extended to regression problems to ($y_k \in \mathbb{R}$ for instance).
The most-intuitive way of classifying an image is to apply a $\mathcal{MLP}$ featured by a neuron per pixel in the input layer. The number of neurons could easily become very large: this choice is not parsimonious. Moreover, in real applications, image features extend over patches that can span several pixels and those patch-wise features do not depend on their position in the image (translational invariance). $\mathcal{MLP}$ do not help neither separating scales (characteristics

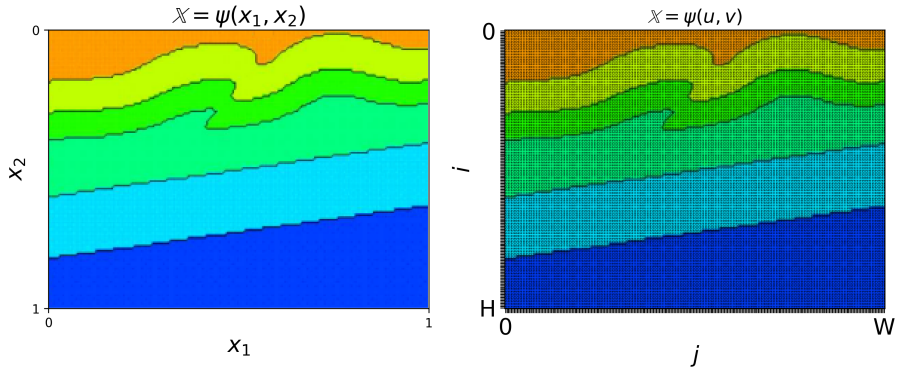


Figure 15: On the left, the `tiff` image $\psi(\boldsymbol{x})$ in RGB scale of geological cross-cut underneath the Kashiwazaki-Kariwa nuclear power plant (see [CGL21]). On the right the digitized discrete image with size $1 \times 3 \times W \times H$ pixels, where the first shape represents number of images (1), the second one the RGB code (3) and the last two the width and height of the image respectively.

features extending over patches of different sizes) nor exploiting possible symmetries and invariant features [Cam20].
A significant advance in this sense was made in 1998 by Y. LeCun and Bengio, whose seminal paper [CBB97] paved the way of designing Convolution Neural Networks $\mathcal{CNN}$, whose widespread usage in real applications kick-started in 2012, due to the work of Krizhevsky et al. [KSH17] that conceived a deep $\mathcal{CNN}$ for the ImageNet LSVRC-2010 classification contest. The latter consisted in developing a 1000-class classifier (see Section 4.1 for further details) based on a database of 1.2 million high-resolution images. Their `AlexNet` achieved top-1 and top-5 error rates of 37.5% and 17.0%, respectively, which at that time was considerably better than the previous state-of-the-art. `AlexNet` is featured by 60 million parameters, arranged in five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers with a final 1000-way $softmax$. This inspired the switch of paradigm: the concept of neurons is replaced by the concept of *convolutional filters*, as sketched in Figure 16, which capture the a priori knowledge of the problem and that considerably reduce the connections of the first layer, thereby facilitating the optimization of the weights by learning.

As shown in Figure 16, each neuron in the first layer (the green spheres in Figure 16) is only connected to (or responsible for) a $k_W \times k_H$ patch of the initial image typically. Each neuron perform a local linear combination of the pixel values which it is connected to and a non-linear activation is applied. In $\mathcal{CNN}$, the weights of each neurons are the same, as if the image was chopped into patches, consecutively fed to the same neuron with $k_W \times k_H \times N_c$ weights.

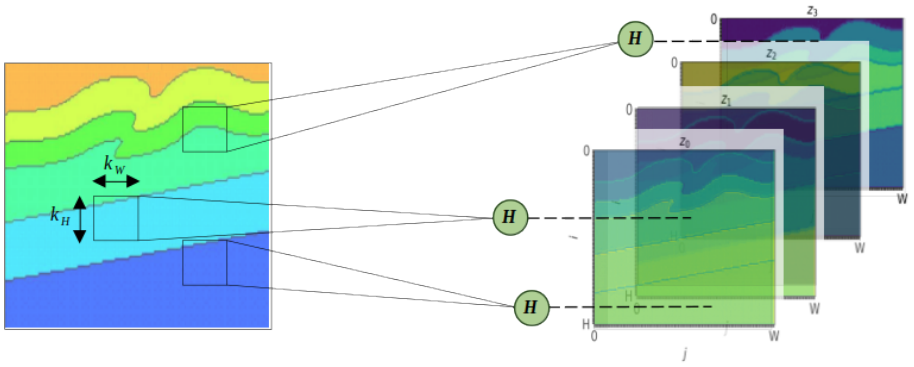


Figure 16: Sketch of convolution layer applied on a the digitized image $\mathbb{X}$ of the geological cross-cut underneath the Kashiwazaki-Kariwa nuclear power plant (see [CGL21]). Four feature maps are shown.

This architecture represents a significant reduction in complexity. N_c represents the number of *filters* outputted by the neuron while the visual patch slides along the image. The neuron weight and bias are independent on the position of the patch, providing a translation-equivariant output.
This invariance by translation is obtained by applying a Linear Translation Invariant (LTI) filter to the image, i.e. a convolution, which $\mathcal{CNN}$ are named from:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^2} \boldsymbol{H}\left(\boldsymbol{u}\right).\boldsymbol{\psi}\left(\boldsymbol{x}-\boldsymbol{u}\right) d\boldsymbol{u} + \boldsymbol{b} = \boldsymbol{H} \star \boldsymbol{\psi}\left(\boldsymbol{x}\right) + \boldsymbol{b} \tag{20}$$

with $\boldsymbol{z}\left(\mathbb{X}\right) = \sum_{c=1}^{N_c} z_c\left(\mathbb{X}\right) \boldsymbol{e}_c$ being called *feature maps* and $\boldsymbol{b}$ the filter bias. Each feature map $z_c\left(\mathbb{X}\right) = \langle \boldsymbol{H} \star \boldsymbol{\psi}, \boldsymbol{e}_c \rangle + b_c$ corresponds to a LTI filter on the image, for a total of N_c filters. The LTI filter in Equation (20) is generically performed over $\mathbb{R}^2$ though the image is defined over the compact support $[0,1]^2$, discretized on the $W \times H$ finite grid. Feature maps are therefore the result of a discrete convolution:

$$\boldsymbol{z}\left[j,i\right] = \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \boldsymbol{H}\left[u,v\right].\boldsymbol{\psi}\left[i-u, j-v\right] + \boldsymbol{b}, \quad (i,j) \in [\![0,W]\!] \times [\![0,H]\!] \tag{21}$$

with $f\left[u,v\right] = f\left(\frac{u}{W}, \frac{v}{H}\right)$ and $\boldsymbol{z}$ being stored in `C` order, contiguous along the rows. In practice, the discrete convolution requires the use of (p_W, p_H) zero-padding in order to make the filter slide along the whole image, as shown in Figure 17.

The filter impulse response $\boldsymbol{H}$ is causal and its discrete counterpart $\boldsymbol{H}\left[u,v\right]$ is defined over the support $[\![0, k_W-1]\!] \times [\![0, k_H-1]\!]$, with $0 < k_W \leq W$ and

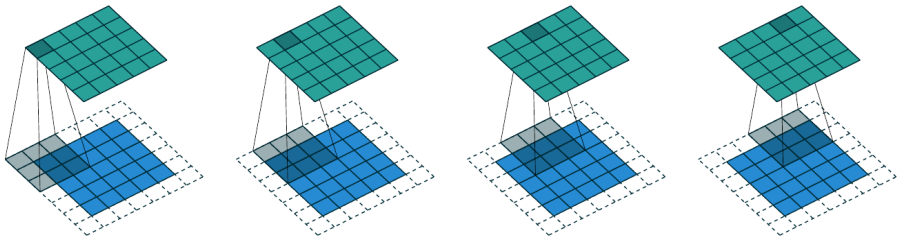


Figure 17: Sketch of the discrete convolution over an image. Reprinted from [DV18] (see https://github.com/vdumoulin/conv_arithmetic). Zero-padding $p_W = p_H = 1$ is added around the image, in order to make the 3×3 kernel slide along the whole image, defined over the $W \times H$ grid.

$0 < k_H \leq H$ (see Equation (203)). If present, the non-linear activation function acts channel-wise, producing feature maps as $\boldsymbol{z}\left(\mathbb{X}\right) = \sum_{c=1}^{N_c} g(z_c\left(\mathbb{X}\right))\boldsymbol{e}_c$.

The convolution can be strided and/or dilated. In its continuous form, the *strided* convolution reads:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^2} \boldsymbol{H}\left(\boldsymbol{u}\right) . \boldsymbol{\psi}\left(\operatorname{diag}\left(s_W, s_H\right) \boldsymbol{x} - \boldsymbol{u}\right) d\boldsymbol{u} + \boldsymbol{b} \tag{22}$$

with strides $s_W \geq 1$ and $s_H \geq 1$. In its discrete counterpart, each strided convolutional filter skips s_W and s_H pixels respectively while sliding along the two directions and it can be written as:

$$\boldsymbol{z}\left[j, i\right] = \sum_{u=0}^{k_W - 1} \sum_{v=0}^{k_H - 1} \boldsymbol{H}\left[u, v\right] . \boldsymbol{\psi}\left[s_W \cdot i - u, s_H \cdot j - v\right] + \boldsymbol{b}, \quad (i, j) \in [\![0, W]\!] \times [\![0, H]\!] \tag{23}$$

The use of stride allows to reduce the spatial dimension of the feature maps by a factor $\frac{1}{s_W}$ and $\frac{1}{s_H}$ respectively, as shown in Figure 18.

Dilating the convolutional filters corresponds to perform an *atrous convolution*, the French "convolution à trous", and it was implemented in $\mathcal{CNN}$ for the first time in [Che+14; YK15]. In other words, dilated convolutions expands the receptive field at a fixed kernel size, by skipping $d_W \geq 1$ and $d_H \geq 1$ pixels between the kernel elements, corresponding to holes in the kernel (see Figure 19).

In its continuous form, dilated convolution is defined by the following expression:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^2} \boldsymbol{H}\left(\operatorname{diag}\left(d_W, d_H\right) \boldsymbol{u}\right) . \boldsymbol{\psi}\left(\boldsymbol{x} - \operatorname{diag}\left(d_W, d_H\right) \boldsymbol{u}\right) d\boldsymbol{u} + \boldsymbol{b} \tag{24}$$

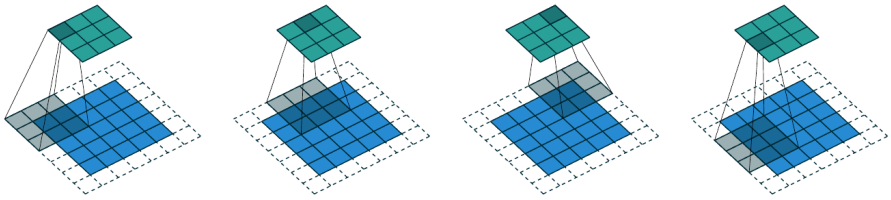


Figure 18: Sketch of the strided discrete convolution over an image. Reprinted from [DV18] (see https://github.com/vdumoulin/conv_arithmetic). Zero-padding $p_W = p_H = 1$ is added around the image, in order to make the 3×3 kernel slide along the whole image, defined over the $W \times H$ grid. The strides $s_W = s_H = 2$ reduce the size of the feature maps by approximately $\frac{1}{s_W}$ and $\frac{1}{s_H}$ respectively (depending on the values of p_W and p_H).

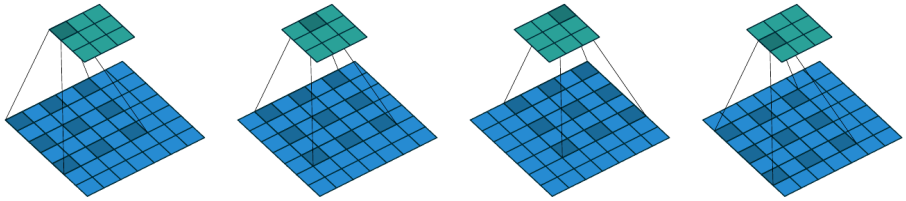


Figure 19: Sketch of the strided discrete convolution over an image. Reprinted from [DV18] (see https://github.com/vdumoulin/conv_arithmetic). The dilation rate $d_W = d_H = 2$ fictitiously widen the kernel size, adding holes in the middle and keeping the same kernel size $k_W = k_H = 3$.

In its discrete form, dilated convolutional filter reads instead:

$$\begin{aligned} \boldsymbol{z}\left[j,i\right] &= \sum_{u=0}^{k_W-1}\sum_{v=0}^{k_H-1} \boldsymbol{H}\left[d_W\cdot u, d_H\cdot v\right].\boldsymbol{\psi}\left[i-d_W\cdot u, j-d_H\cdot v\right], \\ (i,j) &\in [\![0,W]\!]\times[\![0,H]\!] \end{aligned} \tag{25}$$

Equation (25) shows that the dilated convolutions are nothing more than standard convolutions with *effective kernel sizes* $\hat{k}_W = k_W + (k_W - 1)\cdot(d_W - 1)$ and $\hat{k}_H = k_H + (k_H - 1)\cdot(d_H - 1)$.

Example 4. **Convolutional filters in `PyTorch`**
In `PyTorch`, a convolutional layer is defined by the class `Conv2D` as in the following code snippet. Different kernel, stride, padding and dilation rates are adopted, to assess their effect on the image in Figure 15. A hyperbolic tangent activation function is applied to each feature map, in order to compare feature

maps with different kernels, stride, dilation on the same color scale $[-1, 1]$. However, in deep $\mathcal{CNN}$ architectures (see Section 2.6), convolutional filters are first followed by a pixel-wise non-linear activation function and then by a pooling operation.

```
import torch
# assure reproductibility
torch.manual_seed(0)

# Create convolutional feature maps
Nc = 4 # number of output channels
pW = 1 # zero-pad along W
pH = 1 # zero-pad along H
kW = 3 # kernel-size along W
kH = 3 # kernel-size along H
sW = 1 # stride along W
sH = 1 # stride along H
dW = 1 # dilation along W
dH = 1 # dilation along W
p = (pW, pH)
k = (kW, kH)
s = (sW, sH)
d = (dW, dH)
cnn = torch.nn.Sequential(
        torch.nn.Conv2d(in_channels=3, out_channels=Nc,
                        kernel_size=k,
                        stride=s,
                        padding=p),
        torch.nn.Tanh()
        )
# compute feature maps
z = cnn(X).detach().cpu().numpy() # feature maps in numpy format
```

Discrete convolutions are performed on images defined on a grid of $W \times H$ pixels indexed by (i, j). The following algebraic relationship between kernel size, stride, padding, dilation rates (along each dimension) holds [DV18]:

$$\begin{cases} o_i = \lfloor \dfrac{i + 2p_W - k_W - (k_W - 1) \cdot (d_W - 1)}{s_W} \rfloor + 1 & (26) \\ o_j = \lfloor \dfrac{j + 2p_H - k_W - (k_H - 1) \cdot (d_H - 1)}{s_H} \rfloor + 1 & (27) \end{cases}$$

Equation (27) is quite useful to design a convolutional layer: o_i (o_j respectively) represents the output index corresponding to the input index i (j respectively). The following code snippets show how to plot the feature maps:

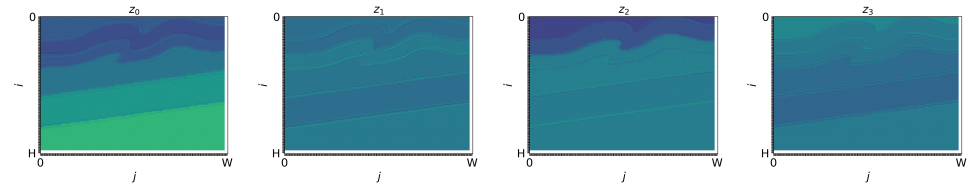
```
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
# Define color map
rainbow_cmap = matplotlib.colormaps['viridis'].resampled(1000)

# loop over the output channels (feature maps)
for c in range(Nc):
    zc = z[0,c,:,:] # c-th feature map

    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(9,4))
```

```
12
13      # plot image
14      ax.imshow(zc, vmin=-1.0, vmax=1.0, cmap= rainbow_cmap)
15
16      # set axes
17      ax.set_xticks(range(0,image.size[0]),
18                    minor=False)
19      ax.set_yticks(range(0,image.size[1]),
20                    minor=False)
21      ax.set_xticklabels([r"0"]+
22          ['' for _ in range(1,image.size[0]-1)]+
23          [r"W"], fontsize=18)
24      ax.set_yticklabels([r"0"]+
25          ['' for _ in range(1,image.size[1]-1)]+
26          [r"H"], fontsize=18)
27      ax.set_xlabel(r"$j$", fontsize=18)
28      ax.set_ylabel(r"$i$", fontsize=18)
29      ax.set_xlim(0,image.size[0])
30      ax.set_ylim(image.size[1],0)
31      ax.set_title(r"$z_{{{:>d}}}$".format(c), fontsize=18)
32
33      # save figure
34      fig.savefig("tsuda_geology_z{}_k{}{}_s{}{}.png".format(c,
35                                                             kW,
36                                                             kH,
37                                                             sW,
38                                                             sH),
39                  dpi=300,
40                  bbox_inches="tight")
```

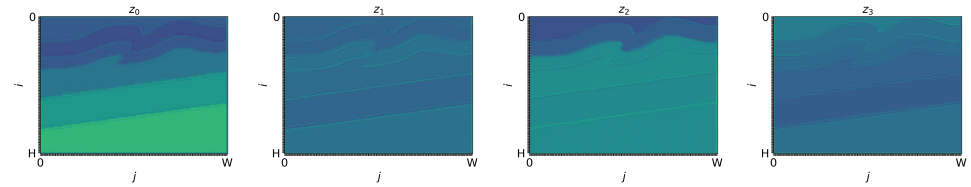
In Figure 20, the effect of two different zero-padding is shown: for $p_W = p_W = 0$ (Figure 20a-Figure 20d) the finite size kernel cannot slide until the end of the image, leaving a narrow blank band at the two bottom-right edges of the images, that consists in an effective downsampling of the image, since the edges are disregarded [MMD20]. This downsampling vanishes by adopting a zero-padding of size $p_W = p_H = 2$ (Figure 20e-Figure 20h). The effect of different kernel sizes $k_W = k_H = 3$ (Figure 21a-Figure 21d) and $k_W = k_H = 5$ (Figure 21e-Figure 21h) is shown in Figure 21. A wider kernel adds extra parameters to the $\mathcal{CNN}$ resulting in a larger receptive field.

Figure 22 shows the effect of changing the stride: $s_W = s_H = 1$ preserves the image size (Figure 22a-Figure 22d), whereas when $s_W = s_H = 2$, the feature maps have half the sizes in both directions (Figure 22e-Figure 22h). Strided convolution reduce the complexity of the initial data, by subsampling it. However, strided convolution do not entail translation-covariant feature maps, with possible loss of the original information contained in the image (see Remark 30). As strided convolutional filters disregard intermediary pixels, translated versions will results in more equivalent outputs if neighbouring pixels are similar to each other across a given patch of the image [MMD20].

Finally, the role of dilation is clear from Figure 23: dilating the convolutional kernel by a rate $d = 5$ (Figure 23e-Figure 23h) widens the receptive field at fixed kernel size, blurring the feature maps by skipping intermediate pixels and adding the blank border if the padding is not changed accordingly. Moreover,

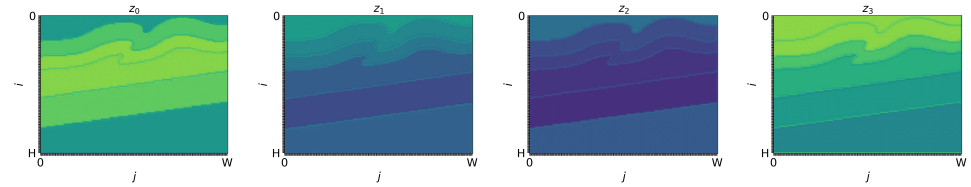


a z_0 with $p = 0$ **b** z_1 with $p = 0$ **c** z_2 with $p = 0$ **d** z_3 with $p = 0$

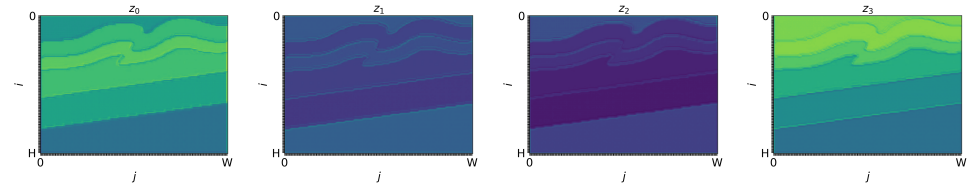


e z_0 with $p = 2$ **f** z_1 with $p = 2$ **g** z_2 with $p = 2$ **h** z_3 with $p = 2$

Figure 20: Effect of the padding for $k_W = k_H = k = 5$, $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ and $p_W = p_H = p = 0$ (a,b,c,d) and $p_W = p_H = p = 2$ for (e,f,g,h).

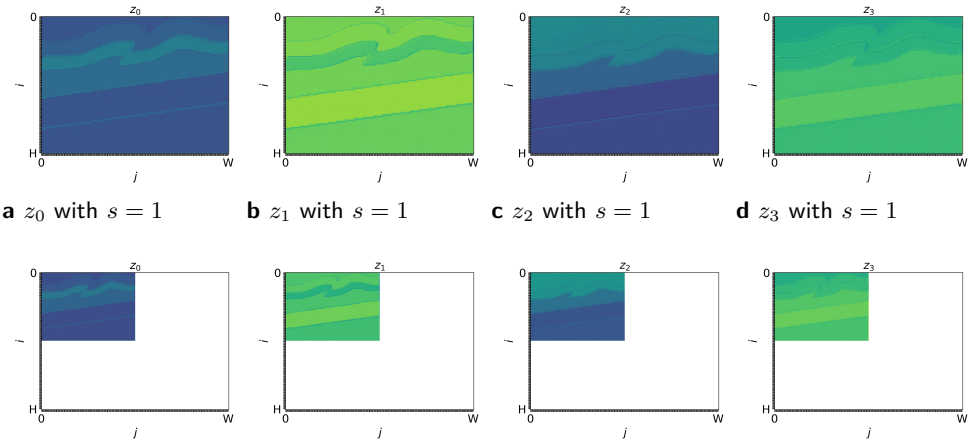


a z_0 with $k = 3$ **b** z_1 with $k = 3$ **c** z_2 with $k = 3$ **d** z_3 with $k = 3$



e z_0 with $k = 5$ **f** z_1 with $k = 5$ **g** z_2 with $k = 5$ **h** z_3 with $k = 5$

Figure 21: Effect of the kernel size for $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ and $k_W = k_H = k = 3$ (a,b,c,d) and $k_W = k_H = k = 5$ for (e,f,g,h). In (a,b,c,d), the padding $p_W = p_H = p = 2$, whereas in (e,f,g,h), $p_W = p_H = p = 1$: this difference in padding does not affect the overall result, except the fact that in both cases the kernel properly slides along the whole image, without leaving blank borders.

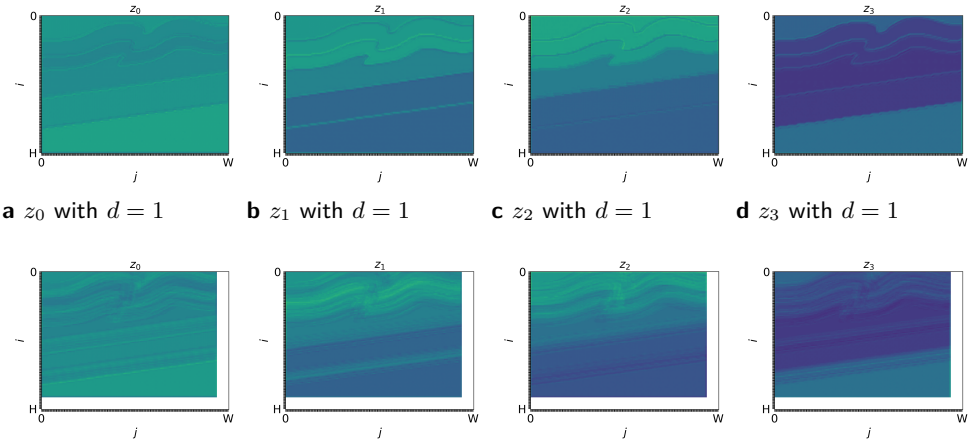


a z_0 with $s = 1$ **b** z_1 with $s = 1$ **c** z_2 with $s = 1$ **d** z_3 with $s = 1$

e z_0 with $s = 2$ **f** z_1 with $s = 2$ **g** z_2 with $s = 2$ **h** z_3 with $s = 2$

Figure 22: Effect of the stride for $k_W = k_H = k = 3$, $p_W = p_H = p = 1$, $d_W = d_H = d = 1$ and $s_W = s_H = s = 1$ (a,b,c,d) and $s_W = s_H = s = 2$ for (e,f,g,h).

the image is "shrunk" by the dilation, with partial information loss (the holes in the fictitious kernel).



a z_0 with $d = 1$ **b** z_1 with $d = 1$ **c** z_2 with $d = 1$ **d** z_3 with $d = 1$

e z_0 with $d = 5$ **f** z_1 with $d = 5$ **g** z_2 with $d = 5$ **h** z_3 with $d = 5$

Figure 23: Effect of the dilation rate for $k_W = k_H = k = 5$, $p_W = p_H = p = 1$, $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ (a,b,c,d) and $d_W = d_H = d = 5$ for (e,f,g,h).

In addition to discrete convolutions, *pooling* operations are often adopted on images. The average- or max-pooling operations also reduce the size of feature maps by outputting either the average or the maximum value of each patch in the image (or in the feature map). Pooling operations are similar to standard convolutional filters, except the fact that the kernel product is replaced by either the average value of the patch or by its maximum value. In particular:

Average pooling ($AvgPooling$) reads (see Section 3.2 fur further insights):

$$\begin{aligned} \boldsymbol{z}\left[j,i\right] &= \frac{1}{k_W \cdot k_H} \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \boldsymbol{\psi}\left[s_W \cdot i + d_W \cdot u, s_H \cdot j + d_H \cdot v\right], \\ & (i,j) \in [\![0,W]\!] \times [\![0,H]\!] \end{aligned} \tag{28}$$

Max-pooling ($MaxPooling$) reads instead:

$$\begin{aligned} \boldsymbol{z}\left[j,i\right] &= \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \max_{u \in [\![0,k_W-1]\!]} \max_{v \in [\![0,k_H-1]\!]} \boldsymbol{\psi}\left[s_W \cdot i + d_W \cdot u, s_H \cdot j + d_H \cdot v\right], \\ & (i,j) \in [\![0,W]\!] \times [\![0,H]\!] \end{aligned} \tag{29}$$

Compared to convolution, pooling operations have no weights nor biases. Moreover, they are channel-wise operations: both Equation (28) and Equation (29) perform average and max operation on each input channel (color of the image or channel of the previous feature map). Therefore, the number of output feature maps is limited to the number of channels fed to the pooling operation.

Example 5. **Comparison between `AvgPool2d`, `MaxPool2d` and `Conv2d` in `PyTorch`**
In `PyTorch`, $AvgPooling$ and $MaxPooling$ on images are two layers defined by the classes `AvgPool2d` and `MaxPool2d`. The code snippet below shows how to set them up, with a hyperbolic tangent activation function.

```
import torch
# assure reproductibility
torch.manual_seed(0)

# Create convolutional feature maps
Nc = 4 # number of output channels
pW = 1 # zero-pad along W
pH = 1 # zero-pad along H
kW = 3 # kernel-size along W
kH = 3 # kernel-size along H
sW = 1 # stride along W
sH = 1 # stride along H
dW = 1 # dilation along W
dH = 1 # dilation along W
p = (pW, pH)
k = (kW, kH)
s = (sW, sH)
d = (dW, dH)
# average pooling
```

```
avgpool = torch.nn.Sequential(
        torch.nn.AvgPool2d(in_channels=3,
                           kernel_size=k,
                           stride=s,
                           padding=p),
        torch.nn.Tanh()
        )
# compute average-pooling feature maps
z_ap = avgpool(X).detach().cpu().numpy() # feature maps in numpy format

# max-pooling
avgpool = torch.nn.Sequential(
        torch.nn.MaxPool2d(in_channels=3,
                           kernel_size=k,
                           stride=s,
                           padding=p),
        torch.nn.Tanh()
        )
# compute average-pooling feature maps
z_mp = avgpool(X).detach().cpu().numpy() # feature maps in numpy format
```

The visual effect of of *AvgPooling* and *MaxPooling* compared to the standard convolution is depicted in Figure 24. The use of hyperbolic tangent as non-linear activation function allows to compare the output of each layer on the same color scale $[-1, 1]$

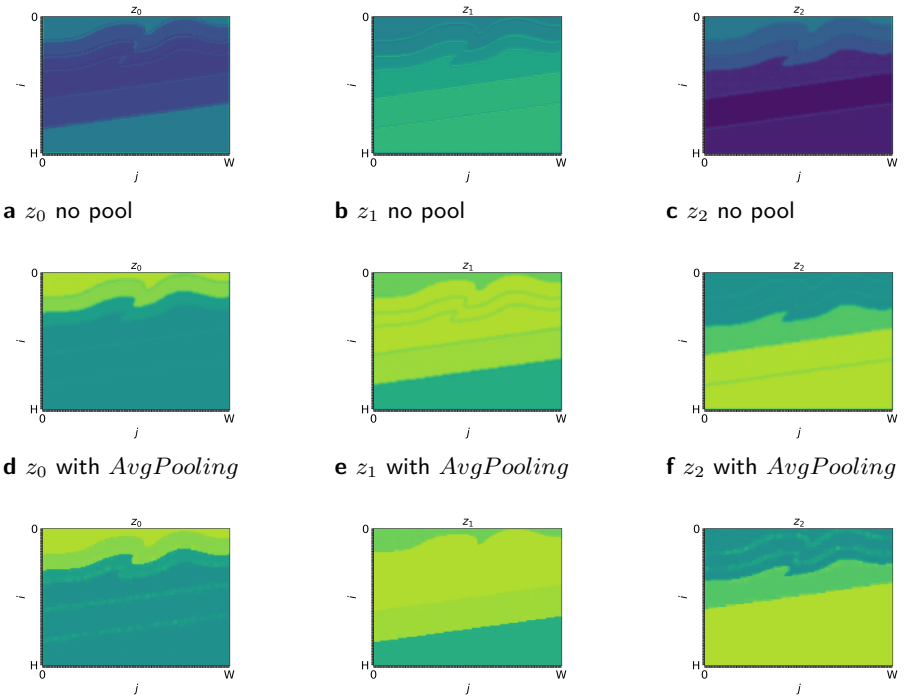
2.6 Deep convolutional architectures

Convolutional layers described in Section 2.5 are generally stacked upon each other, in order to construct deep $\mathcal{CNN}$ such as the one depicted in Figure 25.

Deeper layers act on previous feature maps and the spatial localization in the original image is progressively lost. At the end of a stack of convolution/pooling layers, deep $\mathcal{CNN}$ are featured by some $\mathcal{MLP}$ as output layers, to estimate y. [HS15] estimated that the complexity of a $\mathcal{CNN}$ with N_ℓ convolutional layer, each one with kernel $k_W^{(\ell)} = k_H^{(\ell)} = k^{(\ell)}$ and with no $\mathcal{MLP}$ at the end, is of the order of:

$$\mathcal{O}\left(\sum_{\ell}^{N_\ell} N_c^{(\ell-1)} \cdot N_c^{(\ell)} \cdot k^{(\ell)2} \cdot m^{(\ell)2}\right) \tag{30}$$

with $N_c^{(\ell)}$ being the output channels of each convolutional layer and $m^{(\ell)2}$ the size of the output feature maps at each layer ℓ. The complexity is therefore dominated by $m^{(\ell)2}$, which justify the interest in subsampling, via strided convolution or strided pooling. Subsampling operations drastically reduce the amount of neurons in the last output $\mathcal{MLP}$, leading to even less cumbersome training process. Subsampling has a major downturn though. As a matter of fact, convolutional filters with unitary stride (no subsampling) are translation covariant (see definition in Equation (199)), i.e., the feature maps of a translate images are the translation of the original feature maps. If the original image is subsampled by adopting strided convolution, the similarity between an image and its translated version is lost, because some pixels are skipped as observed in



a z_0 no pool **b** z_1 no pool **c** z_2 no pool

d z_0 with $AvgPooling$ **e** z_1 with $AvgPooling$ **f** z_2 with $AvgPooling$

g z_0 with $MaxPooling$ **h** z_1 with $MaxPooling$ **i** z_2 with $MaxPooling$

Figure 24: Comparison between convolution with $k_W = k_H = k = 3$, $p_W = p_H = p = 2$, $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ (a,b,c), average pooling (d,e,f) and max-pooling (g,h,i).

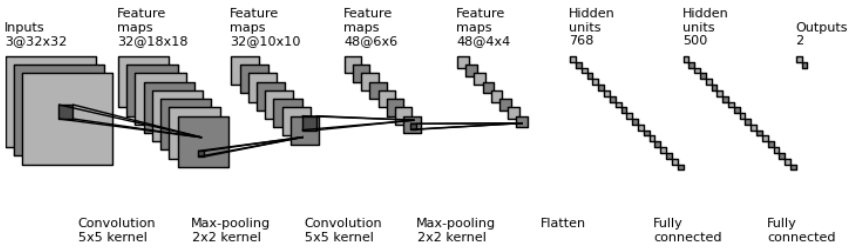


Figure 25: Example of deep $\mathcal{CNN}$ (source: https://github.com/gwding/draw_convnet).

Remark 31 [MMD20]. The remedy consists into use non-srtided convolutions, followed by strided average or max pooling operations, that preserve the local homogeneity of the image, drastically decreasing the size of the output feature and therefore the complexity of the $\mathcal{CNN}$ training scheme. Alternatively, the number of feature maps (*channels*) is generally increased accordingly (stride of 2, double the channels), when strided convolutions are adopted. On the contrary, Average and max pooling grant translation invariance (see Section 3) but they preserve the amount of input channels. The image and successive feature maps can be also shrunk by adopting dilated convolution, or even downsampled by avoiding zero-padding (see Figure 20.
Example 6 shows a simple example of $\mathcal{CNN}$ adopted to classify CIFAR10 database[4], containing 60000 32×32 color images in 10 different classes.

Example 6. **Example of deep convolutional classifier with `PyTorch`**
The following code snippet[5] shows how to easily construct a deep $\mathcal{CNN}$ for CIFAR10 classification.

```
# Source: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# define CNN class as subclass of the master nn.Module
# Once initialized, the forward function must be customized
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels = 3,
                               out_channels = 6,
                               kernel_size = (5,5),
                               stride = (1,1)
                               )

        self.pool = nn.MaxPool2d(kernel_size = 2,
                                 stride = 2)

        self.conv2 = nn.Conv2d(in_channels = 6,
                               out_channels = 16,
                               kernel_size = (5,5),
                               stride = (1,1)
                               )
        self.a1 = nn.Linear(in_features = 16 * 5 * 5,
                            out_features = 120)
        self.a2 = nn.Linear(in_features = 120,
                            out_features = 84)
        self.a3 = nn.Linear(in_features = 84,
                            out_features = 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # conv+relu+pool (1)
        x = self.pool(F.relu(self.conv2(x)))  # conv+relu+pool (2)
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.a1(x)) # linear+relu (1)
        x = F.relu(self.a2(x)) # linear+relu (2)
```

[4] https://en.wikipedia.org/wiki/CIFAR-10#cite_note-1

[5] from `PyTorch` tutorial, see https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

```
        x = self.a3(x) # final output layer
        return x


h_theta = CNN()

# Define loss function
l = nn.CrossEntropyLoss()

# Define SGD optimizer
optimizer = optim.SGD(h_theta.parameters(), lr=0.001, momentum=0.9)
```

The CIFAR10 dataset can be easily accessed via `PyTorch` with the following few lines of code:

```
# Source: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# Define pre-processing operations
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5),
      (0.5, 0.5, 0.5))])

batch_size = 4

# Define train dataset (pre-processing)
Dxy_train = torchvision.datasets.CIFAR10(root='./data',
                                         train=True,
                                         download=True,
                                         transform=transform)
# Define train data loader
Dxy_train = torch.utils.data.DataLoader(Dxy_train,
                                        batch_size=batch_size,
                                        shuffle=True,
                                        num_workers=2)

# Define test dataset (pre-processing)
Dxy_test = torchvision.datasets.CIFAR10(root='./data',
                                        train=False,
                                        download=True,
                                        transform=transform)
# Define test data loader
Dxy_test = torch.utils.data.DataLoader(Dxy_test,
                                       batch_size=batch_size,
                                       shuffle=False,
                                       num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse',
           'ship', 'truck')

# functions to plot an image sample
def imshow(img):
    img = img / 2 + 0.5 # unnormalize
    npimg = img.numpy() # transform to numpy format
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.savefig("CIFAR10_sample.png", dpi=300, bbox_inches="tight")

```

```

# get some random training images
dataiter = iter(Dxy_train)
Xi, yi = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(Xi))
# print labels
print(' '.join(f'{classes[yi[j]]:5s}' for j in range(batch_size)))
```

```
# Source: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
n_e = 100
for epoch in range(n_e):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, batch in enumerate(Dxy_train, 0):
        # get the inputs; data is a list of [inputs, labels]
        Xi, yi = batch

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        h_theta_Xi = h_theta(Xi)
        loss = l(h_theta_Xi, yi)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0
# save NN for future use (inference, fine tuning, ...)
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

# Test on test data

dataiter = iter(Dxy_test)
Xi, yi = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(Xi))
print('GroundTruth: ', ' '.join(f'{classes[yi[j]]:5s}' for j in range(4)))
```

Compared to deep $\mathcal{MLP}$, deep $\mathcal{CNN}$ such as that in Example 6 considerably reduced the initial complexity of the problem because neurons in the convolutional filter maps are specialized and translation covariant and pooling grants the invariance by translation (same class despite its spatial position), overall minimizing the number of weights in each layer [Cam19]. However, deep $\mathcal{CNN}$ can easily reach several hundreds of millions of parameters. The number of parameters is much larger (in general) than the dimensionality of the original data. However, even if the network is trained with thousands or even millions of samples (CIFAR10 is featured by 60000 images), $\mathcal{CNN}$ considerably reduce the dimension of the problem, ranging around 10^6. Typically, the $\mathcal{CNN}$ output layer has $\approx 10^2$ output weights. The complex intermediate representations

$\boldsymbol{z}(\mathbb{X})$ generated by convolutional filters are progressively more and more insensitive to small roto-translations or other symmetries of the image and they linearize the decision boundaries between class (in classification problems).

2.6.1 Transposed convolutions

Transposed convolutions (or "fractionally strided" convolutions) [Zei+10] are the adjoint (transposed) version of the classical convolution defined in Equation (20): in other words, deconvolution. The sketch of the transposed convolution is depicted in Figure 26 In particular, while convolutions are used to en-

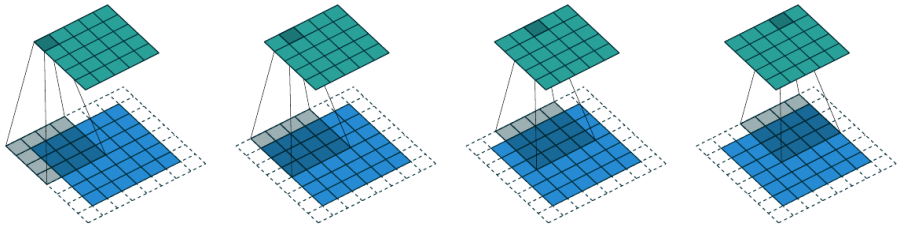


Figure 26: Sketch of the discrete transposed convolution over an image. Reprinted from [DV18] (see `https://github.com/vdumoulin/conv_arithmetic`). Zero-padding $p_W = p_H = 1$ is added around the image, in order to make the 3×3 kernel slide along the whole image, defined over the $W \times H$ grid.

code the data, by projecting it onto a reduced-order latent manifold, transposed convolutions are adopted to decode the latent variables into the data space (see Chapter 3 - *Unsupervised Learning: Basic Concepts and Application to Particle Dynamics* and Chapter 6 - *Non-Euclidean machine learning for geomechanics*). The transposed convolution operation can be thought of as the gradient of some convolution with respect to its input (see Section 5.1), which is usually how transposed convolutions are implemented in practice [DV18]. In `PyTorch`, transposed convolution are designed with the class `ConvTranspose2d`[6].

2.7 Time-forward prediction

$\mathcal{NN}$ have been successfully adopted to ordered flow of information, such as handwriting recognition, speech recognition and in more general to reproduce time-varying physical phenomena. In this sense, the dataset is composed by time series (of signals, of images, of words etc.). In their standard form, feed-forward $\mathcal{NN}$ are fed with fixed-length input data $\boldsymbol{x} \in \mathbb{R}^{d_X}$ and each instance $\boldsymbol{x}_i$ is considered i.i.d. However, the quantities of interest of physical phenomena displaying dynamic behaviour are naturally correlated in time. This aspect can

[6] `https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html`

hardly be captured by feed-forwards $\mathcal{MLP}$ due to their internal structure. In order to feed a time series to a $\mathcal{MLP}$, for regression purposes for instance, the input data stream must be chopped into i.i.d. windows of d_X samples each. However, this represents a quite strong assumption, since long range time correlations could be overlooked. Moreover, trying to feed the $\mathcal{MLP}$ with the whole time-series would entail a major drawback: the number of hidden neurons required to achieve a satisfactory error grows exponentially with the size of the input dimension, as stated in Equation (97):

$$N_K \approx \varepsilon^{\frac{1-d_X}{k}} \tag{31}$$

Finally, the last but not the least reason why the $\mathcal{MLP}$ is not fit for time-series problems is that the time-series itself represent rather irregular functions, i.e., their highest order of (weak) differentiability k is quite low, which implies an even larger number of hidden neurons.

When dealing with discrete time-histories, the data $\mathcal{D}_{XY} = \{(\mathbb{X}_i, \mathbb{y}_i)\}_{i=1}^{N}$ is composed of i.i.d. samples of discrete input signals $\mathbb{X} = \{\boldsymbol{x}[t]\}_{t=1}^{N_t}$ and discrete output signals $\mathbb{y} = \{\boldsymbol{y}[t]\}_{t=1}^{N_t}$, with $\boldsymbol{x}[t] \in \mathbb{R}^{d_X}$ and $\boldsymbol{y}[t] \in \mathbb{R}^{d_Y}$, both of length N_t.

2.8 Recurrent Neural Networks $\mathcal{RNN}$

Recurrent Neural Networks ($\mathcal{RNN}$) [Wer90] are inspired by the $\mathcal{MLP}$ but they are conceived for time-series, since their underlying graph of operation is reflexive. In other words, $\mathcal{RNN}$ is a $\mathcal{NN}$ with one or more feedback loops over time steps [Hay98]. $\mathcal{RNN}$ have infinite impulse response, compared to CNN that have a finite one.

At any time step t, the $\mathcal{RNN}$ produces an output $\boldsymbol{y}[t]$, based on the input time step $\boldsymbol{x}[t]$. The standard $\mathcal{RNN}$ $\boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}[t])$ is defined as follows [GBC16; Sal+18]:

$$\begin{cases} \boldsymbol{a}_t(\boldsymbol{h}, \boldsymbol{x}) = \boldsymbol{W}^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(x)}\boldsymbol{x}[t] + \boldsymbol{b}^{(h)} & (32) \\ \boldsymbol{h}_t(\boldsymbol{a}) = \boldsymbol{g}^{(h)}(\boldsymbol{a}_t) & (33) \\ \boldsymbol{a}_t^{(o)} = \boldsymbol{W}^{(o)}\boldsymbol{h}_t + \boldsymbol{b}^{(o)} & (34) \\ \boldsymbol{z}_t\left(\boldsymbol{a}^{(o)}\right) = \boldsymbol{g}^{(o)}\left(\boldsymbol{a}_t^{(o)}\right) & (35) \\ \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}[t]) = \boldsymbol{z}_t \circ \boldsymbol{h}_t \circ \boldsymbol{a}_t(\cdot, \boldsymbol{h}_{t-1}) \circ \boldsymbol{x}[t] & (36) \end{cases}$$

with $\boldsymbol{h}[t] \in \mathbb{R}^{d_h}$ representing the hidden state states, a sort of "memory" variable that keeps track of past states of the network over many time steps, similarly to what it is usually done in standard auto-regressive models such as ARMA, ARIMA, NARMAX among others[Whi51; Xue+10]. $\boldsymbol{W}^{(h)}$ and $\boldsymbol{b}^{(h)}$ represent the transition weights and biases, associated to the computation of

the hidden state. The value of $\boldsymbol{h}_{t-1}$ in the pre-activation at time t $\boldsymbol{a}_t(\boldsymbol{h}, \boldsymbol{x}) = \boldsymbol{W}^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(x)}\boldsymbol{x}[t] + \boldsymbol{b}^{(h)}$ act as feed-back term. $\boldsymbol{W}^{(o)}$ and $\boldsymbol{b}^{(o)}$ represents weights and biases of the output layer. The biases can be seen as learnable exogenous variables conditioning the time forecast. $g^{(h)}$ and $g^{(o)}$ are the non-linear activation function for hidden layer and output layer. $\boldsymbol{W}^{(h)}$, $\boldsymbol{W}^{(x)}$ are dense weight matrices for hidden and input layers and $\boldsymbol{b}^{(h)}$, $\boldsymbol{b}^{(o)}$ the bias vectors for hidden and output layers respectively. A scheme of the $\mathcal{RNN}$ function is depicted in Figure 27.

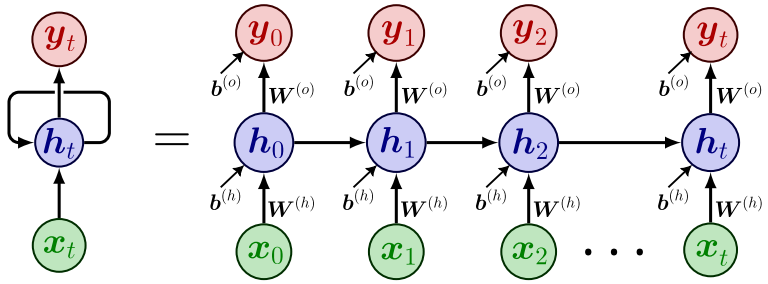


Figure 27: Sketch of $\mathcal{RNN}$ scheme according to Equation (36)

In order to train $\mathcal{RNN}$, the empirical loss is computed as a sum of the empirical losses computed at each time step:

$$L_{\mathcal{D}_{XY}}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{t=1}^{N_t}\sum_{i=1}^{N} l\left(\boldsymbol{y}_i[t], \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}_i[t])\right) \tag{37}$$

It is well known that $\mathcal{RNN}$ are difficult to train [Sal+18] because of their inherent functioning: the long time-span of the dynamics of the hidden state variables, which is certainly necessary to forecast future time steps without the inconvenient of finite impulse response functions (such as in $\mathcal{CNN}$), represents a major cause of training instability. The main reason is the fact the *forward* computational graph is assembled on previous time-steps. For instance, the derivative of $l\left(\boldsymbol{y}_i[t], \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}_i[t])\right)$ with the respect to $W_{ij}^{(x)}$ reads:

$$\frac{\partial l}{\partial W_{ij}^{(x)}}\left(\boldsymbol{y}_i[t], \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}_i[t])\right) = \sum_{k,l,m,n}\frac{\partial l}{\partial z_{t,k}}\frac{\partial g^{(o)}}{\partial h_{t,l}}\frac{\partial g^{(h)}}{\partial a_{t,m}}\left(\frac{\partial a_{t,m}}{\partial h_{t-1,n}}\frac{\partial h_{t-1,n}}{\partial W_{ij}^{(x)}} + \frac{\partial a_{t,m}}{\partial W_{ij}^{(x)}}\right) \tag{38}$$

with the following terms that explicitly read:

$$\begin{cases} \dfrac{\partial g^{(o)}}{\partial h_{t,l}} = \displaystyle\sum_p \frac{\partial g^{(o)}}{\partial a_{t,p}^{(o)}} W_{lp}^{(o)} & (39) \\ \dfrac{\partial a_{t,m}}{\partial h_{t-1,n}} = W_{mn}^{(h)} & (40) \\ \dfrac{\partial a_{t,m}}{\partial W_{ij}^{(x)}} = \delta_{mi} x_j[t-1] & (41) \\ \dfrac{\partial h_{t-1,n}}{\partial W_{ij}^{(x)}} = \displaystyle\sum_{q,r} \frac{\partial g^{(h)}}{\partial a_{t,q}} W_{q,r}^{(h)} \frac{\partial h_{t-2,r}}{\partial W_{ij}^{(x)}} + \frac{\partial g^{(h)}}{\partial a_{t,i}} x_j[t-1] & (42) \end{cases}$$

System 42 proves the fact that the backward propagation unfolds the $\mathcal{RNN}$ in time, i.e., the term $x_j[t-1]$ appears and a recursion on the hidden state variables $\boldsymbol{h}_t$. The gradient of the loss function is the result of chain rule operations that goes back to $\boldsymbol{h}_0$ and it is therefore called Back-Propagation Through Time (BPTT) [Sal+18] that can be generally rewritten, in a more compact way, as follows:

$$\begin{cases} \boldsymbol{\nabla}_{\boldsymbol{W}^{(x)}} l\left(\boldsymbol{y}_i\left[t\right], \boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_i\left[t\right]\right)\right) = \boldsymbol{\nabla}_{\boldsymbol{h}_t} \ell. \left(\displaystyle\sum_{\tau=1}^{t} \boldsymbol{\nabla}_{\boldsymbol{h}_\tau} \boldsymbol{h}_t \otimes \boldsymbol{\nabla}_{\boldsymbol{W}^{(x)}} \boldsymbol{h}_\tau\right) & (43) \\ \boldsymbol{\nabla}_{\boldsymbol{h}_\tau} \boldsymbol{h}_t = \displaystyle\prod_{s=\tau+1} \frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} & (44) \\ \dfrac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} = \boldsymbol{W}^{(h)T} \left(\displaystyle\sum_k \frac{\partial g^{(o)}}{\partial h_{s-1,k}} \boldsymbol{e}_k \otimes \boldsymbol{e}_k\right) & (45) \end{cases}$$

Equation (45) outlines three orders of time dependencies: an "immediate" contribution (i.e., $\boldsymbol{\nabla}_{\boldsymbol{W}^{(x)}} \boldsymbol{h}_\tau$), a "short-term" one and "long-term" one, the latter referring to $\tau \ll t$ [PMB13]. Long-term memory is the main cause of highly instable training processes, since the gradient tends to vanish if the activation functions squash the input (such as σ, $tanh$, etc), as extensively explained in Section 4.2, and the loss at time $t+1$ to increase [Sal+18]. The sufficient condition is that the largest singular value of the recurrent weight matrix $\boldsymbol{W}^{(h)}$ σ_1 must satisfy the condition $\sigma_1 < \frac{1}{\delta}$. In particular, if the diagonal matrix $\frac{\partial g^{(o)}}{\partial h_{s-1,k}} \boldsymbol{e}_k \otimes \boldsymbol{e}_k$ is bounded by above by a term $\gamma \in \mathbb{R}^+$ such as:

$$\left\| \sum_k \frac{\partial g^{(o)}}{\partial h_{s-1,k}} \boldsymbol{e}_k \otimes \boldsymbol{e}_k \right\| \leq \gamma \tag{46}$$

then the Jacobian $\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}}$ is bounded as follows:

$$\left\| \frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} \right\| \leq \left\| \boldsymbol{W}^{(h)} \right\| \cdot \left\| \sum_k \frac{\partial g^{(o)}}{\partial h_{s-1,k}} \boldsymbol{e}_k \otimes \boldsymbol{e}_k \right\| \leq 1 \tag{47}$$

and $\exists \delta \in \mathbb{R}^{+}$ such that $\left\| \frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} \right\| \leq \| \leq \delta < 1$. In this case, the gradient in Equation (45) is bounded as follows:

$$\left\| \boldsymbol{\nabla}_{\boldsymbol{W}^{(x)}} l \left(\boldsymbol{y}_i \left[t \right], \boldsymbol{h}_{\boldsymbol{\theta}} \left(\boldsymbol{x}_i \left[t \right] \right) \right) \right\| = \left\| \boldsymbol{\nabla}_{\boldsymbol{h}_t} \ell . \left(\sum_{\tau=1}^{t} \boldsymbol{\nabla}_{\boldsymbol{h}_\tau} \boldsymbol{h}_t \otimes \boldsymbol{\nabla}_{\boldsymbol{W}^{(x)}} \boldsymbol{h}_\tau \right) \right\| \leq \delta^{t-\tau} \left\| \boldsymbol{\nabla}_{\boldsymbol{h}_t} \ell \right\| \tag{48}$$

The upper bound in Equation (48) shows that BPTT for long time histories tend to make the gradient with the respect to $\tau \ll t$ to vanish whenever $\sigma_1 < \frac{1}{\delta}$, in 5-10 gradient descent epochs (see Section 3 for a detailed description of back-propagation) [PMB13]. Because of its proneness to vanishing gradient, $\mathcal{RNN}$ have major difficulties in capturing dependencies as the duration of dependencies increases [BSF94]. The BPTT can even explode along some direction, if $\sigma_1 > \frac{1}{\delta}$. Vanishing gradient can be counteracted by employing $ReLU$, ELU, $SELU$ and other activation function with non-zero derivative with the respect to their argument, as described and proved in Section 4.2. Exploding gradients are instead counteracted, large batches are considered when using Adam or SGD optimizers, as described in Section 3 and in the example Example 16 that compares different gradient descent algorithms. Alternatively, according to several authors [Wil92; Pér+03; Hay04], $\mathcal{RNN}$ training can highly benefit from Kalman filter application (extended versions), especially for poor datasets, although this strategy is particularly cumbersome. Second order optimization schemes, as described in Section 4.2, such as Newton's methods, are definitely more robust, but still cumbersome, due to the need of computing the Hessian matrix, which in case of a $\mathcal{RNN}$ is prohibitive, due to the time-recursive nature of its formulation. Hessian-free methods, applied to large mini-batches, are rather appealing alternatives to train $\mathcal{RNN}$, especially for non-convex loss function, avoiding the computation of prohibitive Hessian matrices encompassing several time steps, but approximating them to leverage local quadratic approximation in high dimensionality. Another widely used approach to train $\mathcal{RNN}$ is the family of Genetic Algorithms. For deeper insights, a very complete summary of $\mathcal{RNN}$ training techniques can be found in [Sal+18].

Surprisingly, intertwining $\mathcal{MLP}$ and $\mathcal{RNN}$ in deep architectures make the time-forward $\mathcal{NN}$ prediction way more efficient and it reduces the complexity of the training scheme. The cumbersome internal computation that encompasses long-term dependencies can effectively be replaced by shallow feed-forward computations that increase the abstraction level of the input $\boldsymbol{x}\left[t\right]$ time history, with intermediate latent representations $\boldsymbol{\phi}\left(\boldsymbol{x}\right)$, as outlined in Equation (12). In this way, temporal long- and short-time dependencies are better disentangled and the manifolds near which the data concentrate flattened accordingly. $\mathcal{RNN}$ can be combined with $\mathcal{RNN}$ and $\mathcal{MLP}$ according to several combinations, briefly stated in the following paragraphs [Sal+18]:

1. *Deep input-to-hidden*

This configuration corresponds to the $\mathcal{RNN}$ standard formulation in Equation (36), that is *de facto* the combination of two $\mathcal{MLP}$s $\boldsymbol{h}_{FF}^{(h)}$ and $\boldsymbol{h}_{FF}^{(o)}$ respectively are fed to the hidden and output states. $\boldsymbol{x}\,[t]$ is fed to the hidden $\mathcal{MLP}$ that performs the well known following operation:

$$\boldsymbol{h}_{FF}^{(h)}\left(\boldsymbol{x}\,[t]\,,\boldsymbol{h}_{t-1}\right)=$$

$$=\boldsymbol{g}^{(h)}\left(\boldsymbol{a}_t\right)=\boldsymbol{g}^{(h)}\left(\boldsymbol{W}^{(h)}\boldsymbol{h}_{t-1}+\boldsymbol{W}^{(x)}\boldsymbol{x}\,[t]+\boldsymbol{b}^{(h)}\right)$$

Then, the $\mathcal{RNN}$ output is obtained by feeding the output of a second $\mathcal{MLP}$

$$\boldsymbol{h}_{FF}^{(o)}\left(\boldsymbol{h}_t\right)=\boldsymbol{g}^{(o)}\left(\boldsymbol{a}_t^{(o)}\left(\boldsymbol{h}_{FF}^{(h)}\left(\boldsymbol{x}\,[t]\right)\right)\right)=$$

$$=\boldsymbol{g}^{(o)}\left(\boldsymbol{W}^{(o)}\boldsymbol{h}_{FF}^{(h)}\left(\boldsymbol{x}\,[t]\,,\boldsymbol{h}_{t-1}\right)+\boldsymbol{b}^{(o)}\right)$$

To better capture long-term dependencies, the input can be fed to the latter operation too as follows:

$$\boldsymbol{h}_{FF}^{(o)}\left(\boldsymbol{h}_t,\boldsymbol{x}\,[t]\right)=\boldsymbol{g}^{(o)}\left(\boldsymbol{W}^{(o)}\boldsymbol{h}_{FF}^{(h)}\left(\boldsymbol{x}\,[t]\,,\boldsymbol{h}_{t-1}\right)+\boldsymbol{b}^{(o)}+\boldsymbol{W}^{(ox)}\boldsymbol{x}\,[t]+\boldsymbol{W}^{(oh)}\boldsymbol{h}_{t-1}\right)\tag{49}$$

and as shown schematically in Figure 28.

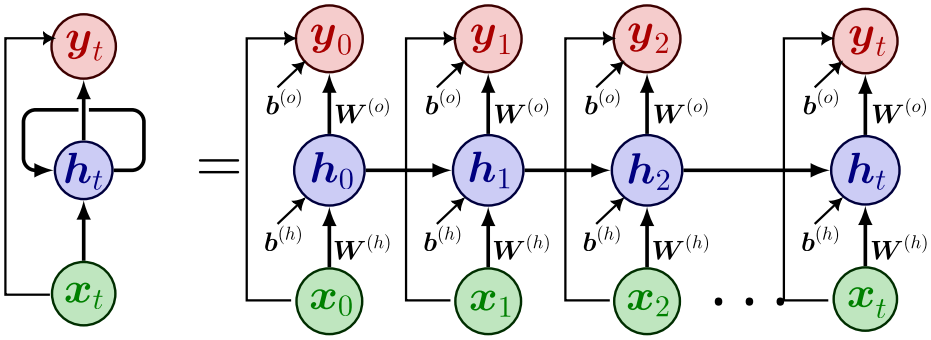


Figure 28: Sketch of the $\mathcal{RNN}$ version with input-to-hidden connection. $\boldsymbol{h}_t^{(1)}$ and $\boldsymbol{h}_t^{(2)}$ represent the two stacked hidden state variables evolving in time and connected to each other. $\boldsymbol{x}_t$ indicates the input time-series, $\boldsymbol{h}_t$ the hidden state and $\boldsymbol{y}_t$ the predicted out time-series.

2. *Deep hidden-to-hidden hidden-to-output*
   $\mathcal{MLP}$ can enhance the level of abstraction of the data time-history $\boldsymbol{x}\,[t]$. leveraging its regularity, with $\mathcal{RNN}$ quickly adjusting to short-time fast changes (for non-stationary signals especially) keeping a good memory

of past events. Further $\mathcal{MLP}$ can be added between hidden and hidden variables, as well as between hidden and output. In the latter case, the $\mathcal{MLP}$ raise the abstraction of the hidden state $\boldsymbol{h}_t$, facilitating the linearization of the decision boundary in its abstract representation and predicting the target in a more robust way. Figure 29 show an example of this architecture.

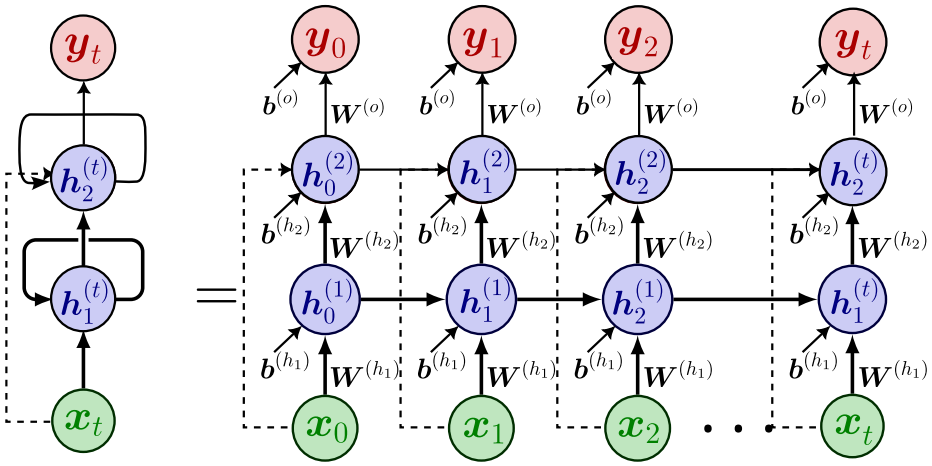


Figure 29: Sketch of the $\mathcal{RNN}$ version with hidden-to-hidden connection. $\boldsymbol{x}_t$ indicates the input time-series, $\boldsymbol{h}_t$ the hidden state and $\boldsymbol{y}_t$ the predicted out time-series.

As byproduct, the hidden state are better summarizing the contribution of the previous inputs [Sal+18].

3. *Stack of hidden states*
   Deep $\mathcal{RNN}$ are easily constructed by stacking recurrent layers, with the effect that each layer is steered to act on different time scales, since each layer receives as input an abstract representation of the original time-history $\boldsymbol{x}[t]$. However, in practice, the transitions between consecutive hidden states is generally limited to a few layers, because deep $\mathcal{RNN}$ tend to be even more prone to vanishing and/or exploding gradient occurrence (as described above), failing into capturing long-term time dependency. Moreover, the overall computational burden can easily become prohibitive because the gradient descent must unravel intricate time-dependency across several layers, with major difficulty in splitting the computation over a parallel architecture (in *model parallel* mode).

2.8.1 Bidirectional $\mathcal{RNN}$

In some context, future time-steps $\boldsymbol{x}[t+\tau]$ need to be accounted for, in a acausal framework. Standard $\mathcal{RNN}$ formulated in Equation (36) could leverage future time-steps by delaying the output of a few steps, but this strategy revealed itself poorly effective. Therefore, *bidirectional* $\mathcal{RNN}$ (or $\mathcal{BRNN}$) were introduced by [SP97], that processes the whole time history back and forth in order to predict the final output. In doing so, two standard $\mathcal{RNN}$ are adopted: one that process $\boldsymbol{x}[t]$ (forward) and the other that processes $\boldsymbol{x}[N_t - t]$ (backward), without reciprocal input/output connection, and hidden states that read [Sal+18]:

$$\begin{cases} \overrightarrow{\boldsymbol{h}}_t = \boldsymbol{g}^{(h)}\left(\overrightarrow{\boldsymbol{W}}^{(h)}\overrightarrow{\boldsymbol{h}}_{t-1} + \overrightarrow{\boldsymbol{W}}^{(x)}\boldsymbol{x}[t] + \overrightarrow{\boldsymbol{b}}^{(h)}\right) & (50) \\ \overrightarrow{\boldsymbol{h}}_t = \boldsymbol{g}^{(h)}\left(\overleftarrow{\boldsymbol{W}}^{(h)}\overleftarrow{\boldsymbol{h}}_t + \overleftarrow{\boldsymbol{W}}^{(x)}\boldsymbol{x}[N_t - t] + \overleftarrow{\boldsymbol{b}}^{(h)}\right) & (51) \\ & (52) \end{cases}$$

The final output is defined as a linear combination of the two:

$$\boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}[t]) = \boldsymbol{g}^{(o)}\left(\overrightarrow{\boldsymbol{W}}^{(o)}\overrightarrow{\boldsymbol{h}}_t + \overleftarrow{\boldsymbol{W}}^{(o)}\overleftarrow{\boldsymbol{h}}_t + \boldsymbol{b}^{(o)}\right) \tag{53}$$

$\mathcal{BRNN}$ (whose architecture is sketched in Figure 30) is slightly more complicated because the update of state and output neurons can no longer be conducted one at a time [SP97]. Another shortcoming is the fact that the time-history must be known from start to end.

Example 7. **Design a $\mathcal{RNN}$ with `PyTorch`**
`PyTorch` has a proper class definition for standard $\mathcal{RNN}$, offering the possibility of adding bi-directionality and stacked layers, as the following code snippet shows.

```
import torch

# design a 2 layer RNN with 50 hidden states (the size of h_t)
d_X = 1 # dimension of the input time history x[t] (in R^1)
N_l = 2 # number of hidden layers
d_h = 50 # dimension of h_t (in R^50)
gactivation = 'tanh' # activation function
bidirectional =False # switch to 'True' for designing a BRNN
bias = True # switch to False if no bias is foreseen
h_theta = torch.nn.RNN(input_size = d_X,
                       hidden_size = d_h,
                       num_layers = N_l,
                       nonlinearity=gactivation,
                       bias=bias,
                       bidirectional=bidirectional)
```

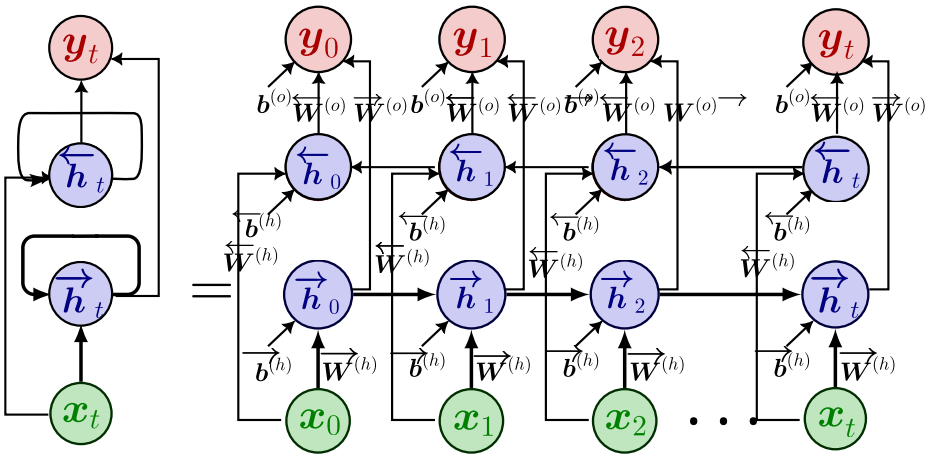


Figure 30: Sketch of the $\mathcal{BRNN}$ version with hidden-to-hidden connections. The left-to-right arrow indicates the weights, biases and operations of the forward $\mathcal{RNN}$, whereas the right-to-left arrow indicates the operations and weights for the backward $\mathcal{RNN}$. $\boldsymbol{x}_t$ indicates the input time-series, $\boldsymbol{h}_t$ (forward and backward respectively) the hidden state and $\boldsymbol{y}_t$ the predicted out time-series.

2.8.2 Combining $\mathcal{RNN}$ and $\mathcal{CNN}$

Recurrent Convolutional Neural Network $\mathcal{RCNN}$ [LH15] leverage the $\mathcal{RNN}$ infinite impulse response, capable of leveraging short- and long-time range memory in the data, with the translational covariance (and equivariance to roto-translation/distortion) of $\mathcal{CNN}$ layers described in Section 2.5 and Section 2.6, capable of tracking time-propagating phenomena regardless their time occurrence. $\mathcal{RNN}$ infinite kernel allows to account for non-local context in the signal, facilitating the classification or regression task to connected $\mathcal{CNN}$, whose aim is to identify the labeling function regardless the time-translation (or spatial translation in an image). $\mathcal{RNN}$ recurrent connections can be intertwined to each convolutional layer in a deep $\mathcal{CNN}$, to enable it with a memory of past information. In this sense, $\mathcal{RNN}$ help stabilizing the classification performance of a deep $\mathcal{CNN}$. Alternatively, a $\mathcal{RNN}$ layer can be stacked at the top of a $\mathcal{CNN}$, whose role is to extract common low-level features from the spatio-temporal phenomenon. The $\mathcal{RNN}$ is fed with the $\mathcal{CNN}$ feature maps, whose high level regularity of abstract representation simplify the $\mathcal{RNN}$ prediction, in a deep input-to-hidden configuration (but replacing the $\mathcal{MLP}$ with a deep $\mathcal{CNN}$).

2.9 Long-Short Term Memory

Hidden variables are cross and delight of standard $\mathcal{RNN}$, since they are irremediably necessary in order to capture long range time dependencies and multi-scale phenomena. However, the long-range dynamics of those hidden features make $\mathcal{RNN}$ very hard to train, since the gradient of the loss function flows backward to the initial time step. Surprisingly, those hidden variables trigger vanishing and exploding gradient events, which make but from a computational point of view, the required memory becomes easily too large to handle. Moreover, despite the possibility of intertwining $\mathcal{MLP}$ and $\mathcal{CNN}$ with recurrent connections in order to simplify the prediction phase thanks to intermediate abstract representation of the input sequence $\boldsymbol{x}[t]$, these hybrid $\mathcal{NN}$ are still hard to define and to tune in order to process short and long-term memory features separately. $\mathcal{RNN}$ do not include any explicit notion of time window, which would be somehow useful for time series prediction/classification. The most popular alternative to $\mathcal{RNN}$ are Long Short-Term Memory recurrent neural network, generically known under the acronym $\mathcal{LSTM}$ [HS97]. The objective of their invention was to introduce a scheme that could improve learning long-range dependencies, by learning to forget [GSC00]. As a matter of fact, the main $\mathcal{LSTM}$ difference is the fact that the $\mathcal{LSTM}$ output is stored in extra "memory cell", with input and output connections controlled by gates g that steer the flow of information from previous time steps (and the gradient back-propagation), as sketched in Figure 31. In particular, the forget gate $g_t^{(f)}$ learns weights that control the rate at which the value stored in the memory cell decays [GSC00].

A standard $\mathcal{LSTM}$ is made of 3 gates: input $\boldsymbol{g}_t^{(i)}$, forget $\boldsymbol{g}_t^{(f)}$, output $\boldsymbol{g}_t^{(o)}$ and a memory cell $\boldsymbol{c}_t$. Gates control the update of the memory cell, preventing its modification for multiple time-steps. This allows $\mathcal{LSTM}$ to train for longer iterations than standard $\mathcal{RNN}$ as well as to capture complex time-scale interdependencies. The equations that rule the standard $\mathcal{LSTM}$ components can be summarized in the following system:

$$
\begin{cases}
\boldsymbol{z}_t^{(i)} = \boldsymbol{g}^{(i)}\left(\boldsymbol{W}^{(ii)}\boldsymbol{x}[t] + \boldsymbol{W}^{(ih)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(ic)}\boldsymbol{g}_t^{(c)} + \boldsymbol{b}^{(i)}\right) & (54)\\
\boldsymbol{z}_t^{(f)} = \boldsymbol{g}^{(f)}\left(\boldsymbol{W}^{(fi)}\boldsymbol{x}[t] + \boldsymbol{W}^{(fh)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(fc)}\boldsymbol{g}_t^{(c)} + \boldsymbol{b}^{(f)}\right) & (55)\\
\boldsymbol{c}_t = \sum_j z_{t,j}(i)\cdot tanh\left(\left\langle \boldsymbol{w}_j^{(ci)}, \boldsymbol{x}[t]\right\rangle + \left\langle \boldsymbol{w}_j^{(ch)}, \boldsymbol{h}_{t-1}\right\rangle + b_j^{(c)}\right)\boldsymbol{e}_j + & (56)\\
+\sum_j z_{t,j}^{(f)}\cdot z_{t-1,j}(c)\boldsymbol{e}_j & (57)\\
\boldsymbol{z}_t^{(o)} = \boldsymbol{g}_t^{(o)}\left(\boldsymbol{W}^{(oi)}\boldsymbol{x}[t] + \boldsymbol{W}^{(oh)}\boldsymbol{h}_{t-1} + \boldsymbol{W}^{(oc)}\boldsymbol{g}_t^{(c)} + \boldsymbol{b}^{(o)}\right) & (58)\\
\boldsymbol{h}_t = \sum_j z_{t,j}^{(o)}\cdot tanh\left(z_{t,j}^{(c)}\right)\boldsymbol{e}_j & (59)
\end{cases}
$$

with:

- $\boldsymbol{W}^{(ii)}$, $\boldsymbol{W}^{(ih)}$, $\boldsymbol{W}^{(ic)}$ and $\boldsymbol{b}^{(i)}$ being the weight matrices and the bias

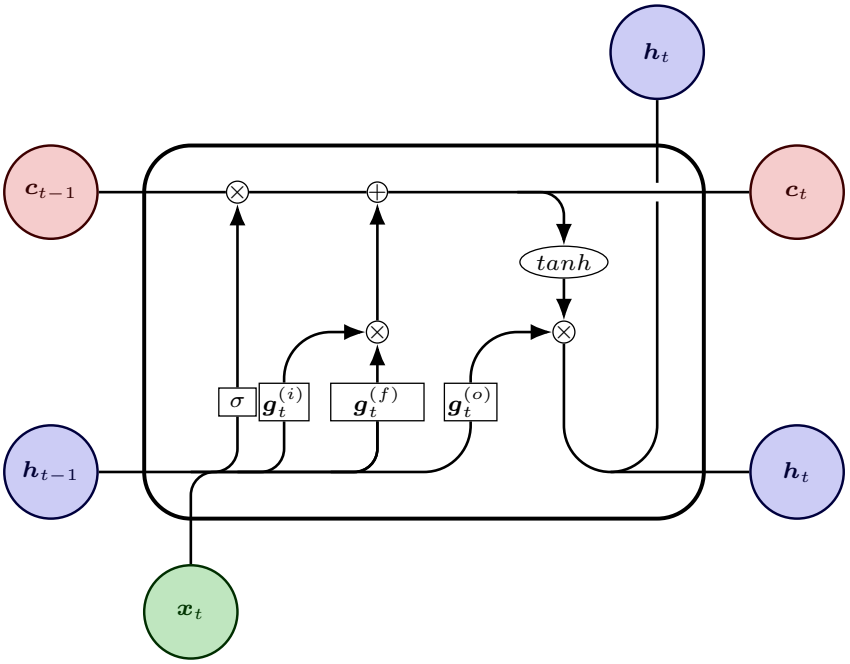


Figure 31: Sketch if the standard $\mathcal{LSTM}$ architecture.

vectors respectively for the input gate, that is represented by a sigmoid activation function applied to each component of the pre-activation, $\boldsymbol{g}_t^{(i)}(\boldsymbol{a}) = \sum_j \sigma(\boldsymbol{a}_j)\boldsymbol{e}_j$;

- $\boldsymbol{W}^{(fi)}$, $\boldsymbol{W}^{(fh)}$, $\boldsymbol{W}^{(fc)}$ and $\boldsymbol{b}^{(f)}$ being the weight matrices and the bias vectors respectively for the forget gate, that is represented by a sigmoid activation function applied to each component of the pre-activation, $\boldsymbol{g}_t^{(f)}(\boldsymbol{a}) = \sum_j \sigma(\boldsymbol{a}_j)\boldsymbol{e}_j$ as per the input gate;

- $\boldsymbol{w}_j^{(ci)}$, $\boldsymbol{w}_j^{(ch)}$, and $b_j^{(f)}$ being the weight matrix j^{th} row and bias vector component respectively for the update of the memory cell $\boldsymbol{c}_t$, that is based on the values of the previously computed input and forget gated activations $\boldsymbol{z}_t^{(i)}$ and $\boldsymbol{z}_t^{(f)}$ respectively;

- $\boldsymbol{W}^{(oi)}$, $\boldsymbol{W}^{(oh)}$, $\boldsymbol{W}^{(oc)}$ and $\boldsymbol{b}^{(o)}$ being the weight matrices and the bias vectors respectively for the output gate, that is represented by a sigmoid activation function applied to each component of the pre-activation,

$\boldsymbol{g}_t^{(o)}(\boldsymbol{a}) = \sum_j \sigma(\boldsymbol{a}_j)\boldsymbol{e}_j$ as per the input and forget gate;

According to [LJH15], Equation (59) explains how the memory cell works. For time-steps t for which input and output gates are deactivated by the sigmoid (negative argument) and for which the forget gate does not apply a higher decay (large positive values of each $z_{t,j}^{(f)}$, that make the sigmoid saturate to 1) the memory cell $\boldsymbol{c}_t$ holds its value $\boldsymbol{z}_{t-1}^{(c)}$, providing a non-vanishing gradient term, which remains constant over those periods. This architecture allows the network to potentially remember information for longer periods, depending on how the forget gates activates. The latter being close to 0 (negative argument of the forget sigmoid), makes the term $\sum_j z_{t,j}^{(f)} \cdot z_{t-1,j}(c)\boldsymbol{e}_j \approx 0$, which means that the network "forgets" the previously store memory cell value. The saturation of the sigmoid activations at input, forget and output gates limit the unbounded growth of the internal variables but this aspect can lead to information loss because there is no active selection of which long-term dependency is relevant or not. Moreover, $\mathcal{LSTM}$ has generally four times the number of parameters of a standard $\mathcal{RNN}$ [Sal+18].

Deep $\mathcal{NN}$ can be constructed by stacking $\mathcal{LSTM}$ hidden layers upon each other, each layer activation defined as:

$$\begin{cases} \boldsymbol{h}_t^{(\ell)} = \boldsymbol{g}^{(\ell)}\left(\boldsymbol{W}^{(ih)}\boldsymbol{h}_t^{(\ell-1)} + \boldsymbol{W}^{(hh)}\boldsymbol{h}_{t-1}^{(\ell)} + \boldsymbol{b}_h^{(\ell)}\right), \quad \forall 1 \leq \ell \leq N_\ell & (60) \\ \boldsymbol{h}_{\boldsymbol{\theta}} = \boldsymbol{g}^{(o)}\left(\boldsymbol{W}^{(o)}\boldsymbol{h}_t^{(N_\ell)} + \boldsymbol{b}^{(o)}\right) & (61) \end{cases}$$

Analogously to bidirection $\mathcal{RNN}$, bidirectional $\mathcal{LSTM}$ have been conceived too for language understanding [GMH13; WJ16]. For a complete summary of all possible $\mathcal{LSTM}$ variants developed in the literature, refer to [Sal+18].

Example 8. **Design a $\mathcal{LSTM}$ with `PyTorch`**
`PyTorch` has a proper class definition for standard $\mathcal{LSTM}$, offering the possibility of adding bi-directionality and stacked layers, as the following code snippet shows.

```
import torch

# design a 2 layer LSTM with 50 hidden states (the size of h_t)
d_X = 1 # dimension of the input time history x[t] (in R^1)
N_l = 2 # number of hidden layers
d_h = 50 # dimension of h_t (in R^50)
gactivation = 'tanh' # activation function
bidirectional =False # switch to 'True' for designing a BRNN
bias = True # switch to False if no bias is foreseen
h_theta = torch.nn.LSTM(input_size = d_X,
                        hidden_size = d_h,
                        num_layers = N_l,
                        nonlinearity=gactivation,
                        bias=bias,
                        bidirectional=bidirectional)
```

Further examples on the use of $\mathcal{RNN}$ and $\mathcal{LSTM}$ will be provided in the dedicated hands-on tutorials.

3 Optimizing a Neural Network

As stated in Section 2, Neural Networks $\mathcal{NN}$ are a special type of statistical models $\mathcal{H}_\theta$, made of complex functions $\boldsymbol{h}_\theta$ defined as composition of non linear ridge activations applied to affine transformations. The composition pattern follows intricate graphs. The search of the "best" set of weights $\boldsymbol{\theta} \in \Theta$ is performed by attempting at minimizing the empirical loss function $L_{\mathcal{D}_{XY}}(\boldsymbol{h}_\theta)$, defined over a set of i.i.d. samples $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, with $\boldsymbol{x}_i \in \mathcal{X} \subset \mathbb{R}^{dx}$ and $\boldsymbol{y}_i \in \mathcal{Y} \subset \mathbb{R}^{d_Y}$ (see problem $(\mathcal{P})$). The quest for the best predictor is a matter of optimization over open sets $\Theta \subset \mathbb{R}^{d\Theta}$, where d_m is the amount of parameters (weights and biases) the $\mathcal{NN}$ is featured of.

Definition 1. Gradient Descent for Empirical Loss Minimization $(\mathcal{G})$
Given an open subset $\Theta \subset \mathbb{R}^{d_m}$, Proposition 76 is a sufficient and necessary conditions for $\hat{\boldsymbol{\theta}}$ to be a minimizer of the Empirical Loss function $L_{\mathcal{D}_{XY}}$ (unless $L_{\mathcal{D}_{XY}}$ is convex, which is not always the case) and it reads:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\arg\min} \quad L_{\mathcal{D}_{XY}}(\boldsymbol{h}_\theta) \tag{62}$$

The quest for $\hat{\boldsymbol{\theta}}$ is iteratively conducted, following the direction of $-\boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}$, according to the following update rule from iteration i to iteration $+i+1$ (the so called *delta rule*:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta^{(i)} \boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right), \quad \eta^{(i)} \in \mathbb{R}^+ \tag{63}$$

with $\eta^{(i)} \in \mathbb{R}^+$ being the *learning rate*, and

$$L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right) \geq L_{\mathcal{D}_{XY}}\left(\hat{\boldsymbol{\theta}}\right)$$

Remark 2. The solution of problem $(\mathcal{P})$ is obtained by computing the gradient of the empirical loss function (see Equation (1)), which reads:

$$\boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{D}_{XY}} \boldsymbol{\nabla}_\theta \ell\left(\boldsymbol{h}_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i\right) \tag{64}$$

Some further theoretical details are provided in Appendix B. Moreover, Section 4.1 outlines the theoreteical background that justify the strategies to minimize $L_{\mathcal{D}_{XY}}$ presented in the following.

Remark 3. The cost of computing the exact gradient in Equation (64), by averaging across the whole dataset instances and then update the weights (the so

called Batch Gradient Descent) is $O(N \cdot d_m)$. This computational cost is reasonably affordable for rather small dataset and shallow $\mathcal{NN}$, being tailored only for convex and rather smooth loss functions. However, for very large dataset, the floating-point error induced by computing a sum over a large number of addends introduces spurious errors that can make the gradient descent algorithm to fail.

Remark 4. The standard gradient descent practice employs single-precision floating-point format (i.e., numbers are encoded in a 32-bit-long (4 bytes) binary code) and even half-precision floating-point format (i.e., numbers are encoded in a 16-bit-long (2 bytes) binary code). The choice of employing simple- or even half-precision floating-point format is justified by the fact that gradient descent algorithms are run on Graphic Processing Units (GPU), primarily designed for interactive rendering, not computation. Compared to Central Processing Units (CPUs) GPUs have limited cache memory (where to store data to be repeatedly accessed) but higher sequential memory (where to store data that are naturally structured in a contiguous way, such as in an array). Therefore, a GPU is extremely fast and efficient in computing linear algebra operations on contiguously stored data, that the CPU is in charge of feeding it continuously. However, to reach peak performances, those linear algebra operations (such as the gradient descent one) are performed on single- or even half-precision floating-point format, so to be executed faster. Double-precision floating-point format can hinder the GPU performances, due to its limited memory capacity, which is why it is rarely used. However, as observed in Remark 3, this choice implies possible divergence of the Batch Gradient Descent algorithm on large datasets.

3.1 Stochastic Gradient Descent (SGD)

Inspired by the Fisher's approach for maximizing the log-likelihood function[7] with unbiased estimator, a suitable minimization strategy would be to pursue the minimum via the delta rule in Equation (63) via successive random gradient updates (see) computed from uniformly sampled database $\mathcal{D}_{XY}$ according to the following algorithm, called Stochastic Gradient Descent (SGD):

In other words, the SGD approximates the gradient in Equation (64) by randomly sampling the all the i.i.d. samples over different epochs (n_e is the epoch's number). $\boldsymbol{\theta}^{(i+1)}$ are random vectors that converge towards $\hat{\boldsymbol{\theta}}$ at a certain speed. SGD effectively reduces the computational burden mentiond in Remark 3, since its complexity is estimated to $O(d_m)$, which makes it an appealing alternative to the classical GD alternative. The choice of the learning rate scheduling is

[7] or minimizing the negative log-likelihood which belongs to the family of empirical loss functions

Algorithm 1 Stochastic Gradient Descent and Batch Gradient Descent

$i = 0$
Initialize $\boldsymbol{\theta}^{(0)}$
$\eta^{(0)}$
while $i < n_e$ **do**
  for $t \sim \mathcal{U}\left(\{i\}_{i=1}^{N}\right)$ **do**
    $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta^{(i)} \boldsymbol{\nabla}_{\theta} \ell \left(\boldsymbol{h}_{\theta}\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}^{(i)} \right), \quad \eta^{(i)} \in \mathbb{R}^{+}$
  end for
end while

usually made in the following way [Pey20]:

$$\eta^{(i)} = \frac{\eta^{(0)}}{1 + \frac{i}{i_0}}, \quad i_0 > 0 \tag{65}$$

or, more generally:

$$\eta^{(i)} = \frac{\eta^{(0)}}{1 + i\eta_d} \tag{66}$$

The learning rate update scheme in Equation (65) ensures that $\eta^{(i)} \leq \eta^{(0)}$ and, after a "warm up" phase, it decreases as the epochs go by. The value of η_d represents the so called learning rate decay. However, other scheduling strategies are available, such as Cyclic Learning Rate Scheduler [Smi17]. The latter suggests to first calibrate the lower and upper bounds $\eta_{\min}$ and $\eta_{\max}$ respectively (if not known before hand, such as shown in Section 4.3), and then reiterates several loops such as shown in Section 3.1. [Smi17] seemingly suggests that 1 cycle could be enough.

Within the framework of the SGD algorithm, the delta rule in Equation (63) is usually applied by averaging the randomly sampled gradients on "mini-batches" of N_b instances each, i.e., adopting the following algorithm: Averaging on mini-batches is certainly cost-effective compared to standard SGD, that corresponds to N mini-batches of $N_b = 1$ instance each. SGD on mini-batches can be seen as a vectorized version of the SGD.

Example 9. **Stochastic Gradient Descent for linear regression**

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from matplotlib import pyplot as plt
# Data for regression problem
A = 10.1542550
b = 14.7351129
c = 2.34
```

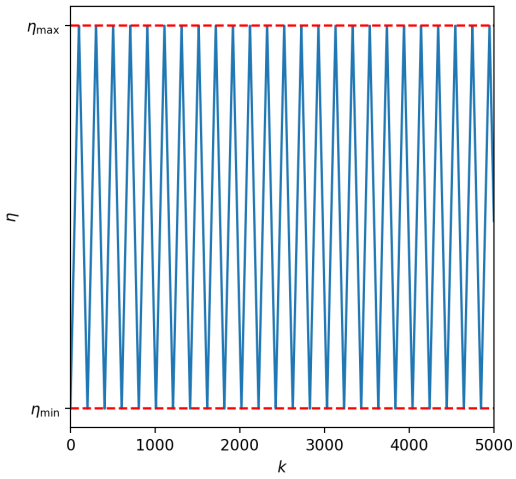


Figure 32: Example of Cyclic Learning Rate Scheduler [Smi17].

```
epsilon = 10.154988
N = 200 # number i.i.d. samples
Nb = 10 # number of samples in the mini-batch
# Data Generation
np.random.seed(42)
x = np.random.rand(N, 1)
y = b + A * np.exp(c*x) + epsilon * np.random.randn(N, 1)

# Shuffled indices
idx = np.arange(N)
np.random.shuffle(idx)

Ntrain = int(N*0.8)
# Uses first 80% of the dataset for training
train_idx = idx[:Ntrain]
# Uses the last 20% for validation
val_idx = idx[Ntrain:]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]

# Convert Numpy arrays into PyTorch Tensors
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

# Plot the dataset
# plt.scatter(x,y,color='black',label=r"$D_{XZ}$")
# plt.scatter(x[train_idx],
#             y[train_idx],
#             color='red',
#             marker="+",
#             label="train dataset")
# plt.scatter(x[val_idx],
#             y[val_idx],
#             color='blue',
#             marker="x",
```

Algorithm 2 Stochastic Gradient Descent on mini-batches

$i = 0$
Initialize $\boldsymbol{\theta}^{(0)}$
$\eta^{(0)} = \eta_0$
while $i < n_e$ **do**
  for $j \sim \mathcal{U}\left(\{i\}_{i=1}^{\frac{N}{N_b}}\right)$ **do**
    $\boldsymbol{g}_\theta = 0$
    for $t \sim \mathcal{U}\left(\{j_i\}_{i=1}^{N_b}\right)$ **do**
      $\boldsymbol{g}_\theta += \boldsymbol{\nabla}_\theta \ell\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}^{(i)}\right)$
    end for
    $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \frac{\eta^{(i)}}{N_b}\boldsymbol{g}_\theta, \quad \eta^{(i)} \in \mathbb{R}^+$
  end for
end while

```
#             label="validation dataset")
# plt.xlabel(r"$x$")
# plt.ylabel(r"$y$")
# plt.legend(frameon=False)

eta = 1e-1 # constant learning rate
# for warmup: https://github.com/Tony-Y/pytorch_warmup
n_e = 1000 # number of epochs

# Creating a MLP model with 1 layer and 1 neuron
h_theta = nn.Sequential(nn.Linear(1, 1), nn.SiLU())

# Define the SGD optimizer
optimizer = optim.SGD(h_theta.parameters(), lr=eta)
L_Dxy = nn.MSELoss()

# Loop over the training epochs on training dataset
i=0
while i < n_e: # loop over epochs
  for j in range(0,int(Ntrain/Nb)): # loop over mini-batches
    t_idx = np.arange(j*Nb,(j+1)*Nb,1) # mini-batch linear indexing
    np.random.shuffle(t_idx) # shuffle mini-batch
    x_b = x_train_tensor[t_idx]
    y_b = y_train_tensor[t_idx]
    optimizer.zero_grad() # initialize the optimizer
    loss = 0.0 # initialize the loss
    for t in range(Nb): # loop over the samples in the batch
      xt = x_b[t]
      yt = y_b[t]
      yhat = h_theta(xt) # predict the output
      loss += L_Dxy(yhat, yt) # Compute Empirical Loss
    loss.backward() # Compute derivatives with AutoGrad
    optimizer.step() # weight update!
```

3.2 Beyond SGD: the role of "momentum"

The SGD in Algorithm 1 and its mini-batch version in Algorithm 2 are first-order Hessian-free methods that converges slowly and it is sensitive to noise [Sut+13; Pey20]. Moreover, according to Remark 34, the convergence can be hindered when the Hessian is ill-conditioned. A possible improvement is represented by the so called "momentum", using previous gradient memory. The Classical Momentum technique [Pol64] accelerates the gradient descent, with a momentum coefficient $\gamma \in [0, 1)]$ that cumulates a velocity vector $\boldsymbol{v}$ (or gradient memory) in directions of persistent reduction in $L_{\mathcal{D}_{XY}}$ across iterations [Sut+13]. Algorithms 3 and 4 outline the SGD algorithm with Classical Momentum, without and with mini-batch average.

Algorithm 3 Stochastic Gradient Descent with Classical Momentum

$i = 0$
Initialize $\boldsymbol{\theta}^{(0)}$
$\eta^{(0)} = \eta_0$
$\boldsymbol{v}^{(0)} = \mathbf{0}$
while $i < n_e$ **do**
  for $t \sim \mathcal{U}\left(\{i\}_{i=1}^{N}\right)$ **do**
    $\boldsymbol{v}^{(i+1)} = \eta^{(i)} \left(1 - \langle \tau^{(i)} - 1 \rangle\right) \boldsymbol{\nabla}_{\theta} \ell \left(\boldsymbol{h}_{\theta}\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}^{(i)}\right) + \gamma^{(i)} \boldsymbol{v}^{(i)}, \quad \eta^{(i)} \in \mathbb{R}^{+}, \gamma^{(i)} \in [0, 1)$
    $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \boldsymbol{v}^{(i+1)}$
  end for
end while

The factor $-\langle \tau^{(i)} - 1 \rangle$ dampens the gradient only if $\tau^{(i)} > 1$[8]. When no dampening is considered (i.e. $\tau^{(i)} \leq 1$), the Classical Momentum accelerates by a (usually) constant factor γ (often chosen equal to 0.85 or 0.95) the gradient descent in the directions $\boldsymbol{e}_{\xi}$ is small but persistent. For instance, in the mini-batch version in Algorithm 4, if $\left\langle \boldsymbol{g}_{\theta}, \boldsymbol{e}_{\xi} \right\rangle > 0$

$$\left\langle \boldsymbol{e}_{\xi}, \boldsymbol{v}^{(i)} \right\rangle = \eta^{(i)} \left\langle \boldsymbol{e}_{\xi}, \boldsymbol{g}_{\theta} \right\rangle + \gamma^{(i)} \left\langle \boldsymbol{e}_{\xi}, \boldsymbol{v}^{(i-1)} \right\rangle \geq \gamma^{(i)} \left\langle \boldsymbol{e}_{\xi}, \boldsymbol{v}^{(i-1)} \right\rangle \tag{67}$$

The inertia allows to overcome local minima and saddle points but, on the contrary, its effect vanishes along the directions of small or oscillating reduction in the objective loss across iterations are amplified by Algorithm 3. For convex functions, momentum-based methods are known to outperform SGD in the early or transient stages [Sut+13]. [Pol64] proved that the Classical Momentum algorithm accelerates convergence to local minima by reducing by a factor $\sqrt{\kappa\left(\mathbf{H}_{L_{\mathcal{D}_{XY}}}\right)}$ the number of iterations required by greedy algorithms

[8] $\langle \cdot \rangle$ represent the Macaulay brackets.

Algorithm 4 Stochastic Gradient Descent on mini-batches with Classical Momentum

$i = 0$
Initialize $\boldsymbol{\theta}^{(0)}$
$\eta^{(0)} = \eta_0$
$\boldsymbol{v}^{(0)} = \mathbf{0}$
while $i < n_e$ **do**
  for $j \sim \mathcal{U}\left(\{i\}_{i=1}^{\frac{N}{N_b}}\right)$ **do**
    $\boldsymbol{g}_\theta = 0$
    for $t \sim \mathcal{U}\left(\{j_i\}_{i=1}^{N_b}\right)$ **do**
      $\boldsymbol{g}_\theta += \boldsymbol{\nabla}_\theta \ell\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}^{(i)}\right)$
    end for
    $\boldsymbol{v}^{(i+1)} = \eta^{(i)}\left(1 - \langle \tau^{(i)} - 1 \rangle\right)\boldsymbol{g}_\theta + \gamma^{(i)}\boldsymbol{v}^{(i)}, \quad \eta^{(i)} \in \mathbb{R}^+, \gamma^{(i)} \in [0, 1)$
    $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \frac{\eta^{(i)}}{N_b}\boldsymbol{v}^{(i+1)}, \quad \eta^{(i)} \in \mathbb{R}^+$
  end for
end while

such as the one in Equation (152). In this case, γ must be chosen equal to $\frac{\sqrt{\kappa\left(\mathbf{H}_{L_{\mathcal{D}_{XY}}}\right)}-1}{\sqrt{\kappa\left(\mathbf{H}_{L_{\mathcal{D}_{XY}}}\right)}+1}$, similar to the expression in Equation (172). Second-order methods, as seen in Section 4.2, amplify those steps in low-curvature directions , but instead of accumulating changes they reweight the update along each eigen-direction of the curvature matrix by the inverse of the associated curvature [Sut+13]. On the contrary, Classical Momentum algorithms can miss the minimizer, due to the consistent inertia of the gradient descent.
In `PyTorch`, the Classical Momentum is an option of the standard `SGD` optimizer.

Example 10. **Stochastic Gradient Descent for linear regression with Classical Momentum**
This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
...
# Define the SGD optimizer
gamma = 0.5/eta # Classical Momentum algorithm
# PyTorch adopts a Classical Momentum coefficient eta*mu
# which corresponds to gamma in the above mentioned formulas
tau = 0.0 # Dampening coefficient
optimizer = optim.SGD(h_theta.parameters(), lr=eta,
        momentum=gamma, dampening=tau)
...
```

3.3 Beyond Classical Momentum SGD: the Nesterov algorithm

A better convergence rate than the Classical Momentum algorithm is provided by the Nesterov algorithm, especially for general smooth (non-strongly) convex functions and a deterministic gradient descent [Sut+13]. The difference with the Classical Momentum method resides in the fact that the gradient of the loss function is computed at the position updated with $\boldsymbol{v}^{(i)}$. The Nesterov Accelerated Gradient algorithm reads: Usually, $\gamma^{(i)}$=0.9. In `PyTorch`, the Nesterov

Algorithm 5 Nesterov Accelerated Gradient Descent

$i = 0$
Initialize $\boldsymbol{\theta}^{(0)}$
$\eta^{(0)} = \eta_0$
$\boldsymbol{v}^{(0)} = \mathbf{0}$
while $i < n_e$ **do**
  for $t \sim \mathcal{U}\left(\{i\}_{i=1}^{N}\right)$ **do**
    $\boldsymbol{v}^{(i+1)} = \eta^{(i)}\left(1 - \langle \tau^{(i)} - 1 \rangle\right) \boldsymbol{\nabla}_\theta \ell\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}^{(i)} - \gamma^{(i)} \boldsymbol{v}_i\right) + \gamma^{(i)} \boldsymbol{v}^{(i)}, \eta^{(i)} \in \mathbb{R}^+, \gamma^{(i)} \in [0, 1)$
    $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i+1)} - \boldsymbol{v}^{(i)}$
  end for
end while

Algorithm 6 Nesterov Accelerated Gradient Descent on mini-batches

$i = 0$
Initialize $\boldsymbol{\theta}^{(0)}$
$\eta^{(0)} = \eta_0$
$\boldsymbol{v}^{(0)} = \mathbf{0}$
while $i < n_e$ **do**
  for $j \sim \mathcal{U}\left(\{i\}_{i=1}^{\frac{N}{N_b}}\right)$ **do**
    $\boldsymbol{g}_\theta = 0$
    for $t \sim \mathcal{U}\left(\{j_i\}_{i=1}^{N_b}\right)$ **do**
      $\boldsymbol{g}_\theta += \boldsymbol{\nabla}_\theta \ell\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}_i - \gamma^{(i)} \boldsymbol{v}^{(i)}\right)$
    end for
    $\boldsymbol{v}^{(i+1)} = \eta^{(i)}\left(1 - \langle \tau^{(i)} - 1 \rangle\right) \boldsymbol{g}_\theta + \gamma^{(i)} \boldsymbol{v}^{(i)}, \quad \eta^{(i)} \in \mathbb{R}^+, \gamma^{(i)} \in [0, 1)$
    $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \frac{\eta^{(i)}}{N_b} \boldsymbol{g}_\theta, \quad \eta^{(i)} \in \mathbb{R}^+$
  end for
end while

Accelerated Gradient algorithm is an option of the standard `SGD` optimizer.

Example 11. **Nesterov Accelerated Gradient Descent for linear regression**
This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
...
# Define the SGD optimizer
gamma = 0.9 # Nesterov momentum coefficient
tau = 0.0 # Dampening coefficient
optimizer = optim.SGD(h_theta.parameters(), lr=eta,
        nesterov=True, momentum=gamma)
...
```

3.4 Optimizing with adaptive learning rates

Several strategies have been developed to improve the convergence rate of first-order Hessian-free gradient descent algorithms. Those algorithms are based on adaptive schemes of the learning rate. In the following, the most popular ones are presented. For the sake of simplicity, we consider a deterministic Gradient Descent algorithm

3.4.1 Adaptive Gradient: AdaGrad

[DHS11] proposed an adaptive update rule called *AdaGrad*. The latter is based on the definition of $\mathbb{G}^{(i)}$, defined as:

$$\mathbb{G}^{(i)} = \sum_{k=0}^{i-1} \boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(k)}\right) \otimes \boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(k)}\right) \tag{68}$$

The AdaGrad update rule reads:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \left(\text{diag}\mathbb{G}^{(i)} + \epsilon \mathbb{I}\right)^{-\frac{1}{2}} . \boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right) \tag{69}$$

with usual values of η=10^{-2} and a numeric tolerance ϵ=10^{-8}. The diagonal matrix diag($\mathbb{G}$) represents the cumulated square derivatives along θ_j, $G_j^{(i)} = \sum_{k=0}^{i-1}\left(\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_j}\left(\boldsymbol{\theta}^{(k)}\right)\right)^2$ from epoch 0 to epoch i. Moreover, $\sqrt{\text{Tr}\left(\mathbb{G}^{(i)}\right)}$ represents the ℓ_2 norm of $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_j}$, but with each gradient component associated to one iteration. For $i \gg$, $\frac{G_j^{(i)}}{i-1}$ is an unbiased estimator of the variance $\mathbb{E}\left[\left(\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_j}\left(\boldsymbol{\theta}_i\right)\right)^2\right]$ and it is monotonically increasing. AdaGrad requires to save and update diag$\mathbb{G}^{(i)}$ at each iteration. The effective learning rate of each gradient component is expressed as $\eta_j^{'} = \frac{\eta}{G_j^{(i)}+\epsilon}$. $\eta_j^{'}$ can become very small, due to the monotonic increase of $G_j^{(i)}$, until the update stops. This aggressive

learning rate decay represents one major shortcoming of the AdaGrad. In order to mitigate such effect, AdaGrad update can be combined with the adaptive learning rate decay in Equation (66).

Example 12. **AdaGrad for linear regression**
This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
...
# Define the AdaGrad optimizer
etad = 0.0 # learning rate decay
epsilon = 1e-8 # tolerance
G0 = 0.0
optimizer = optim.Adagrad(h_theta.parameters(), lr=eta,
                          lr_decay=etad, eps=epsilon, initial_accumulator_value=G0)
...
```

3.4.2 Root Mean Square Propagation: RMSprop

Another optimization strategy, capable of mitigating the disadvantage of aggressive learning rate decay, proper to AdaGrad, was proposed by [TH12] and called *RMSprop*. The idea is to compute an average value of $\mathbb{G}^{(i)}$, based on a shorter memory window, i.e.:

$$\text{diag}\tilde{\mathbb{G}}^{(i)} = \alpha_r \ \text{diag}\mathbb{G}^{(i-1)} + (1-\alpha_r)\left(\text{diag}\mathbb{G}^{(i)} - \text{diag}\mathbb{G}^{(i-1)}\right) \tag{70}$$

The update rule is the same as Equation (69), but with $\text{diag}\tilde{\mathbb{G}}^{(i)}$ to replace $\text{diag}\mathbb{G}^{(i)}$ and usually the smoothing constant α_r=0.9. The main difference is represented by the fact that $\text{diag}\tilde{\mathbb{G}}^{(i)}$ is not monotonically increasing, therefore the learning rate decay is slower.

Example 13. **RMSprop for linear regression**
This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
...
# Define the RMSprop optimizer
alphar = 0.9 # smoothing constant (called alpha in PyTorch)
epsilon = 1e-8 # tolerance
G0 = 0.0
optimizer = optim.RMSprop(h_theta.parameters(), lr=eta,
                          alpha=alphar, eps=epsilon)
...
```

3.4.3 RMSprop with momentum: ADADELTA

The RMSprop with momentum (see Section 3.2) is called *ADADELTA* [Zei12]. In particular, after having computed $\tilde{\mathbb{G}}^{(i)}$ defined in Equation (70), ADADELTA

computes the vector $\boldsymbol{U}^{(i+1)}$ with momentum, i.e., as follows:

$$\boldsymbol{U}^{(i+1)} = \alpha_u \boldsymbol{U}^{(i)} + (1 - \alpha_u)\, \boldsymbol{u}^{(i)} \tag{71}$$

with $\boldsymbol{u}^{(i)}$ that slightly modifies the gradient with a term approximating the inverse Hessian matrix (diagonal), but based on previous gradient update, according to the expression:

$$\boldsymbol{u}^{(i+1)} = \text{diag}^{\frac{1}{2}} \left(\mathbb{U}^{(i)} + \epsilon \mathbb{I} \right) \text{diag}^{-\frac{1}{2}} \left(\tilde{\mathbb{G}}^{(i)} + \epsilon \mathbb{I} \right) . \boldsymbol{\nabla}_{\theta} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right) \tag{72}$$

where $\mathbb{U}^{(i)} = \left(\boldsymbol{U}^{(i)} \otimes \boldsymbol{U}^{(i)} \right)^{\frac{1}{2}}$. All the vectors $\boldsymbol{u}^{(i+1)}$ has the same dimensions of $\boldsymbol{\theta}$. The term $\text{diag}^{\frac{1}{2}} \left(\mathbb{U}^{(i)} + \epsilon \mathbb{I} \right)$ represents the inverse Hessian approximation (see Equation (155)), with the contribution of the RMSprop update, whereas the term $\text{diag}^{-\frac{1}{2}} \left(\tilde{\mathbb{G}}^{(i)} + \epsilon \mathbb{I} \right)$ the contribution of the RMSprop gradient. Finally, $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \boldsymbol{u}^{(i)}$. The advantage of the ADADELTA is that it is parameter-free, since no learning rate *stricto sensu* is required. This prevents the possibility of a learning rate scheduler. Another ADADELTA's shortcoming is represented by the biased momentum estimate in Equation (71) and Equation (72), strictly depending on the weights' initialization.

Example 14. **Adadelta for linear regression**
This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
...
# Define the ADADELTA optimizer
alphau = 0.9 # smoothing constant (called rho in PyTorch)
epsilon = 1e-8 # tolerance
optimizer = optim.Adadelta(h_theta.parameters(), lr=eta,
                           rho=alphau, eps=epsilon)
...
```

3.4.4 Adaptive Moment Estimation: Adam

The Adaptive Moment Estimation, *Adam*, invented by [KB15], is a rather famous optimizer in deep learning. Adam retrieves the idea of momentum, by estimating the first (mean) and second order (variance) moments. Adam endorse the idea of momentum by keeping a moving average of previous gradients (first moment) which will be more or less important depending on the value of a coefficient β_1. Adam is quite similar to ADADELTA (see Section 3.4.3), but it adds another value (second moment) which will be the sliding average of the previous squared gradients whose square root corresponds to the sliding uncentered variance of the previous gradients. Adam adopts the following steps:

- It estimates the diagonal of Hessian as done in ADADELTA by the term $\text{diag}^{\frac{1}{2}}\left(\mathbb{U}^{(i)}+\epsilon\mathbb{I}\right)\text{diag}^{-\frac{1}{2}}\left(\tilde{\mathbb{G}}^{(i)}+\epsilon\mathbb{I}\right)$ but with local accumulation:

$$\text{diag}\mathbb{V}^{(i)}=\beta_2\mathbb{V}^{(i-1)}+(1-\beta_2)\,\text{diag}\mathbb{G}^{(i)} \tag{73}$$

- It estimates the first-order gradient moment, similarly to $\boldsymbol{u}^{(i+1)}$ in Equation (71):

$$\boldsymbol{m}^{(i)}=\beta_1\boldsymbol{m}^{(i-1)}+(1-\beta_1)\,\boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right) \tag{74}$$

- It normalizes both $\text{diag}\mathbb{V}^{(i)}$ and $\boldsymbol{m}^{(i)}$ to avoid to produce biased estimators:

$$\hat{\boldsymbol{m}}^{(i)}=\frac{\boldsymbol{m}^{(i)}}{1-\beta_1},\qquad \hat{\mathbb{V}}^{(i)}=\frac{\mathbb{V}^{(i)}}{1-\beta_2} \tag{75}$$

  This additional operation reduces the biases of the first iterations where the moving averages are made on a small number of values.

- It updates the weights as:

$$\boldsymbol{\theta}^{(i+1)}=\boldsymbol{\theta}^{(i)}-\eta\cdot\text{diag}^{-\frac{1}{2}}\left(\hat{\mathbb{V}}^{(i)}+\epsilon\mathbb{I}\right)\hat{\boldsymbol{m}}^{(i)} \tag{76}$$

Adam has several advantages: the per-dimension update inherited from ADADELTA, which is very important to promote and to tackle sparsity. As RMSprop and ADADELTA, it adopts the use of second order momentum on the denominator. This aspect is tightly linked to the information of the statistical model represented by $\mathcal{NN}$ parametrized by its weights and biases $\boldsymbol{\theta}$. As a matter of fact, the Cramér-Rao bound (CRB) presented in Equation (31) provides an insightful view since it states that the Fisher information $\mathbb{I}_F(\boldsymbol{\theta})$ is larger the the inverse variance of the the weights (represented by $\text{diag}^{-\frac{1}{2}}\hat{\mathbb{V}}^{(i)}$). In other words, the optimizer tries to increase the overall informative power of the $\mathcal{NN}$, to make it more flexible to complex regression or classification tasks. Moreover, Adam has the advantage of integrating momentum, but also another parameter that will prevent the momentum from taking too much importance in updating the weights. These two tools will allow it is up to Adam to dampen the effects of oscillation in the gradient descent.

Example 15. **Adam for linear regression**
This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
...
# Define the SGD optimizer
beta1 = 0.9 # Adam coefficients
beta2 = 0.999 # Adam coefficients
epsilon = 1e-8 # tolerance
optimizer = optim.Adam(h_theta.parameters(), lr=eta,
                       betas=(beta1, beta2), eps=epsilon)
...
```

3.4.5 Weight decay

The weight decay allows to keep $\|\boldsymbol{\theta}\|$ small. Heuristically, this is a preferable condition to achieve convergence in the optimization process. Compared to explicitly penalize the L^2 norm in the loss function, by adding something of the sort $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|^2$, the weight decays makes the weights tend to zero quite faster. The weight decay consists in simply adding an extra term $\lambda\boldsymbol{\omega}^{(i)}$ either to the gradient $\boldsymbol{\nabla}_\theta L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right)$, either directly to the weight update. In this latter case, the weight decay is decoupled from first and second order moments. For instance, with the Adam optimizer, the decoupled weight decay reads:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \cdot \operatorname{diag}^{-\frac{1}{2}}\left(\hat{\mathbb{V}}^{(i)} + \epsilon\mathbb{I}\right)\hat{\boldsymbol{m}}^{(i)} + \lambda\boldsymbol{\omega}^{(i+1)} \tag{77}$$

The `PyTorch optimizer` class implements a `weight_decay` option. This extra parameter can directly added to the `SGD` optimizer input list, since the `SGD` does not estimate neither the first nor the second gradient moments. On the contrary, `RMSprop`, `Adagrad`, `AdaDelta`, `Adam` do. However, adding the `weight_decay` parameter to those standard optimizer classes, couples the weight decay with the estimation of the first and second moments, which is not the most effective way to perform weight decay, since it can add spurious biases. Instead, in order to apply a decoupled weight decay as in Equation (77), one should use the `SGDW` or `AdamW` optimizers[9].

Example 16. **Compare optimizers in `PyTorch`**
In the following example [Via22], the `SGD` optimizer, with and without (Nesterov) momentum is compared to `Adam`, `Adadelta`, `RMSProp` and `AdamW`, in the framework of an optimization problem with $\boldsymbol{\theta} \in \mathbb{R}^2$ and with the following loss function:

$$\begin{aligned} L(\theta_1, \theta_2) = {} & 8 \cdot \sin\left(\frac{1.5 \cdot \theta_1}{10} - 3\right) \cdot \sin\left(\frac{1.5 \cdot \theta_1}{10} - 3\right) + \\ & + 8 \cdot \cos\left(\frac{\theta_2}{10} - 2\right) \cdot \cos\left(\frac{\theta_2}{10} - 2\right) + \\ & 0.3 \cdot \left(\left(\frac{\theta_1}{5} - 12\right) \cdot \left(\frac{\theta_1}{5} - 12\right) + \left(\frac{\theta_2}{10} - 8\right) \cdot \left(\frac{\theta_2}{10} - 8\right)\right) + 10 \end{aligned} \tag{78}$$

$L(\boldsymbol{\theta})$ is depicted in Figure 33.

The comparison is performed according to following criteria:

1. Parameter initialization $(\theta_1^{(0)}, \theta_2^{(0)})$ (see Section 4.2.1 for further insights).
2. Use of the (Nesterov) momentum and dampening (for `SGD`) and `RMSprop`.
3. Weight decay for Adam and AdamW.

[9] https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html

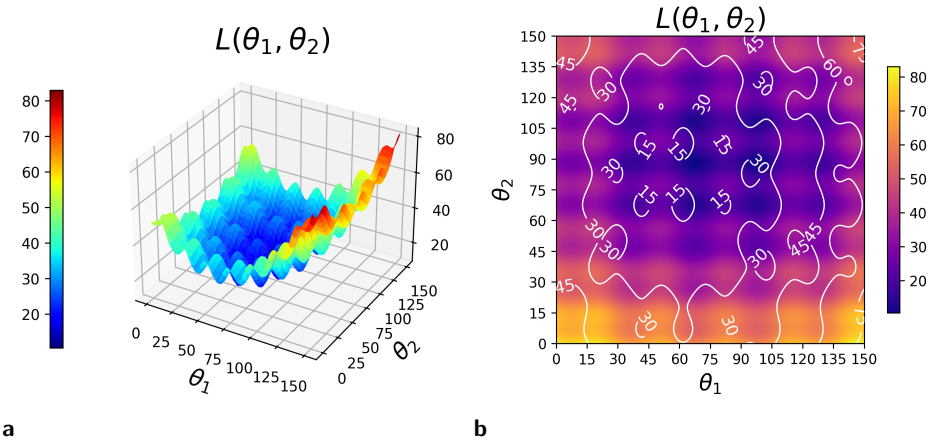


Figure 33: $L(\boldsymbol{\theta}) : \mathbb{R}^2 \to \mathbb{R}$ defined in Equation (78). (a) $L(\boldsymbol{\theta})$ in 3 dimensions. (b) 2-dimensional contour plot.

All tested optimizers share the same (constant) learning rate η=20 and 10000 epochs.
First and foremost, the effect of two different initializations $\boldsymbol{\theta}^{(0-1)}$=(135,135) and $\boldsymbol{\theta}^{(0-2)}$=(45,35) is tested. The choice of $\boldsymbol{\theta}^{(0-1)}$ implies that the optimizer algorithm starts from a point closer to the minimum $\hat{\boldsymbol{\theta}}$ than $\boldsymbol{\theta}^{(0-2)}$ (see Figure 33). Figure 34 shows the `SGD`'s high sensitivity to the initialization: the algorithm diverges from $\hat{\boldsymbol{\theta}}$ (within the 10000 epochs) when starting from $\boldsymbol{\theta}^{(0-2)}$ (see Figure 34b), which is closer to the minimum itself. This is mostly due to the fact that the function is not convex close to the minimum $\hat{\boldsymbol{\theta}}$.

`Adadelta` (with $\alpha_u \in [0, 1]$, called `rho` in `PyTorch`) fails for both initialization choices, as depicted in Figure 35. This is due to its high dependency on the weight initialization with biased momentum estimate.

`Adam` (with standard choice of $\beta_1 = 0.9$ and $\beta_2 = 0.999$) is definitely more robust than `SGD` and more effective than `Adadelta`, as shown in Figure 36, although the trend is opposite: `Adam` converges to $\hat{\boldsymbol{\theta}}$ for $\boldsymbol{\theta}^{(0-2)}$ (as shown in Figure 36b) and it gets somehow closer to it than `SGD`, for $\boldsymbol{\theta}^{(0-1)}$, as depicted in Example 16, compared to Figure 34a. Compared to standard `SGD`, `Adam` leverages the first and second order momenta and adapts their contribution to avoid divergence of the gradient descent algorithm. As a matter of fact, `SGD` needs (first order) momentum too, in order to converge to the minium in a reasonable amount of epochs (10000), as depicted in Figure 34a. This is evident if one compares the latter to the trajectory that `SGD` with zero momentum follows, depicted in Figure 37a (with τ=0.5 in order to avoid divergence). The lack of momentum

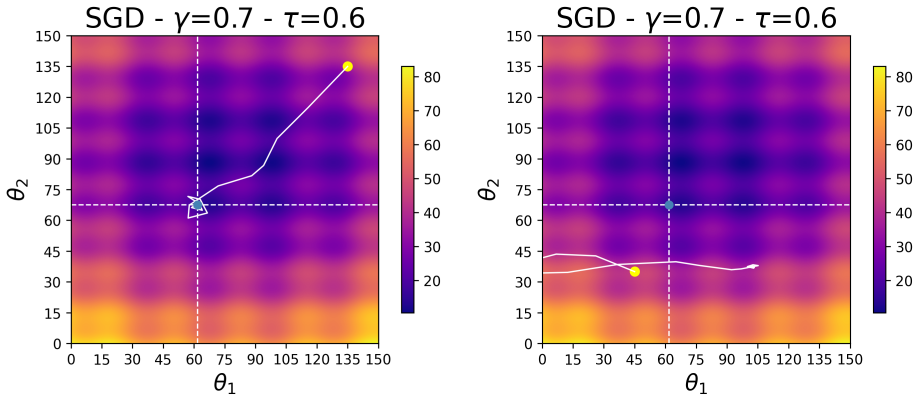


Figure 34: `SGD` minimization trajectory for $\boldsymbol{\theta}^{(0-1)}$ (a) and $\boldsymbol{\theta}^{(0-2)}$ (b). `SGD` is featured by momentum γ=0.7 and dampening τ=0.6. The yellow dot represents the starting point $\boldsymbol{\theta}^{(0)}$, whereas the blue dot represents the sought minimum.

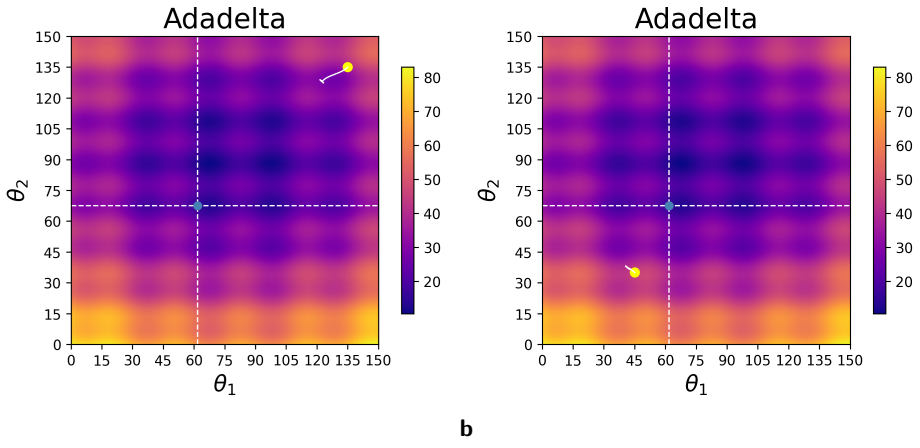


Figure 35: `Adadelta` minimization trajectory for $\boldsymbol{\theta}^{(0-1)}$ (a) and $\boldsymbol{\theta}^{(0-2)}$ (b). `Adadelta` is featured by α_u=0.9, but similar results are obtained regardless its value $\alpha_u \in [0, 1]$. The yellow dot represents the starting point $\boldsymbol{\theta}^{(0)}$, whereas the blue dot represents the sought minimum.

leaves the `SGD` algorithm iterating around local minima. However, a too high momentum γ can lead the `SGD` algorithm to miss $\hat{\boldsymbol{\theta}}$ because of its inertia, as shown in Figure 37b for γ=0.9. For practical purposes, this aspect represents

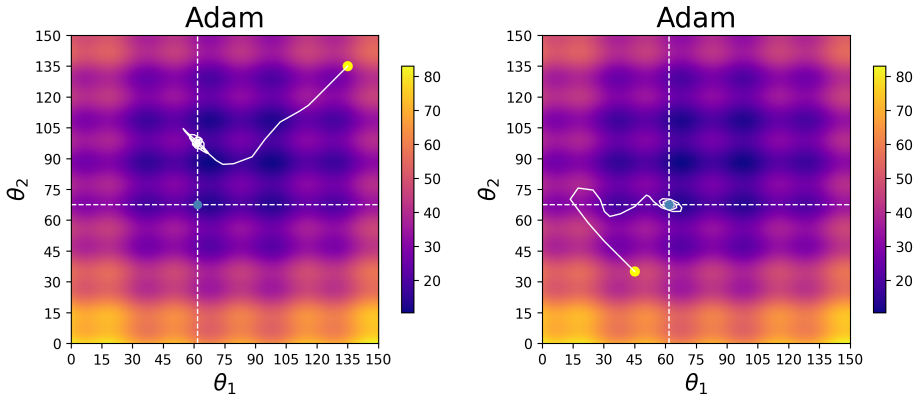


Figure 36: `Adam` minimization trajectory for $\boldsymbol{\theta}^{(0-1)}$ (a) and $\boldsymbol{\theta}^{(0-2)}$ (b). The yellow dot represents the starting point $\boldsymbol{\theta}^{(0)}$, whereas the blue dot represents the sought minimum.

the major disadvantage of using `SGD`, compared to `Adam` adaptivity and faster convergence rate.

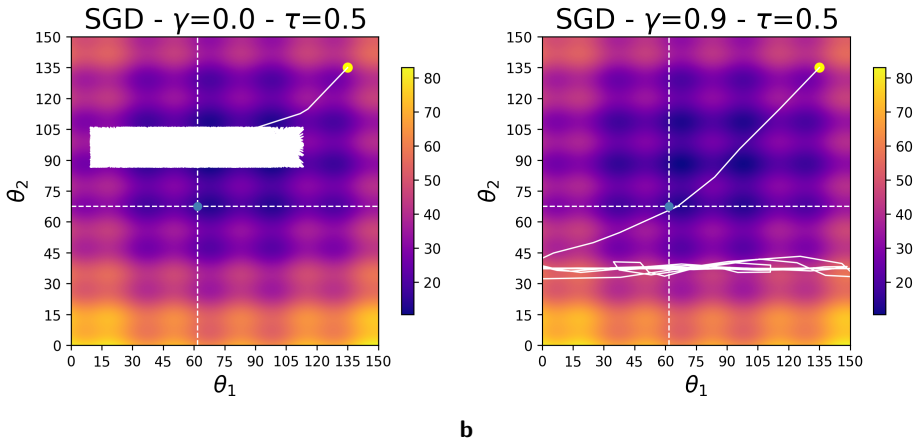


Figure 37: `SGD` minimization trajectory for γ=0 and τ=0.5 (a) and for γ=0.9 and τ=0.5. The yellow dot represents the starting point $\boldsymbol{\theta}^{(0-1)}$, whereas the blue dot represents the sought minimum.

Nesterov momentum enhance the momentum effect (no damping is allowed in this algorithm), possibly leading `SGD` to reach the minimum $\hat{\boldsymbol{\theta}}$ faster (in less

epochs). However, as shown in Figure 38 for different initializations $\boldsymbol{\theta}^{(0-1)}$ (in Figure 38a) and $\boldsymbol{\theta}^{(0-2)}$ (in Figure 38b), Nesterov-`SGD` algorithm tends to range around the minimum $\hat{\boldsymbol{\theta}}$, but missing it because of the inertia effect (each final step represents a too large increment, that iteratively misses the minimum). In this case, a proper schedule of the learning rate must be adopted. `RMSprop` is also conceived to counteract aggressive weight updates, based on

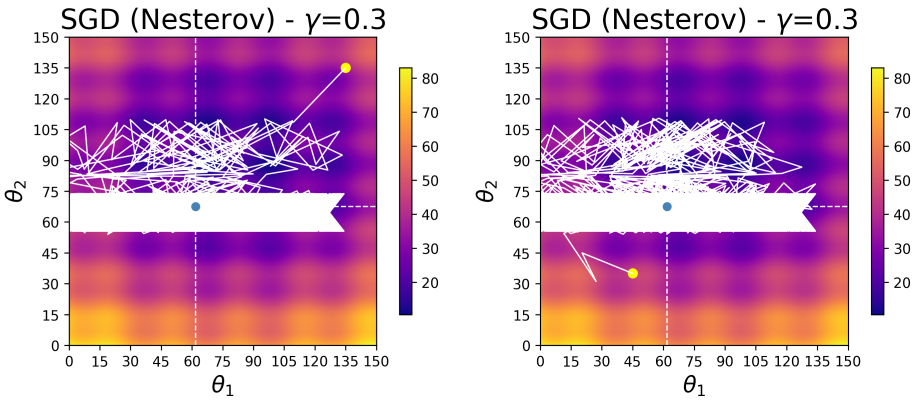


Figure 38: `SGD` minimization trajectory for $\boldsymbol{\theta}^{(0-1)}$ (a) and $\boldsymbol{\theta}^{(0-2)}$ (b). `SGD` is featured by Nesterov momentum γ=0.3 and no dampening. The yellow dot represents the starting point $\boldsymbol{\theta}^{(0)}$, whereas the blue dot represents the sought minimum.

the estimated variance of the gradients computed in past iterations. Despite being more stable and effective that `SGD` with (Nesterov) momentum, for both weight initializations, in this example `RMSprop` achieve a "stable" gradient descent trajectory only when the momentum γ is compensated by the coefficient α_r and both are equal to 0.5 (see Figure 39b). However, [Mni+16] showed that RMSprop is quite useful for very non-stationary problems, as confirmed by [ACB17], provided that no momentum is added (γ=0), α_r=0.9 and the learning rate is very small. Reducing the learning rate to 12, the minimum is reached with a clear improvement in the stability of the gradient descent trajectories, for both initializations, as shown in Figure 40. However, the instability close to the minimum $\hat{\boldsymbol{\theta}}$ still persists. From those examples, it looks that calibrating `SGD`, `Adadelta`, `RMSprop` in order to find a reasonable trade-off between convergence rate and stability is hard: either a small learning rate is assumed, with zero momentum and a very large number of epochs demanded to reach the minimum, either the dampened momentum is leverage, with high risk of divergence. Standard `Adam`, on the contrary, seems definitely more robust and flexible to complex optimization tasks. As far as weight decay is concerned, `Adam` (no weight decay) and `AdamW` (with weight decay) are compared in Figure 41. The

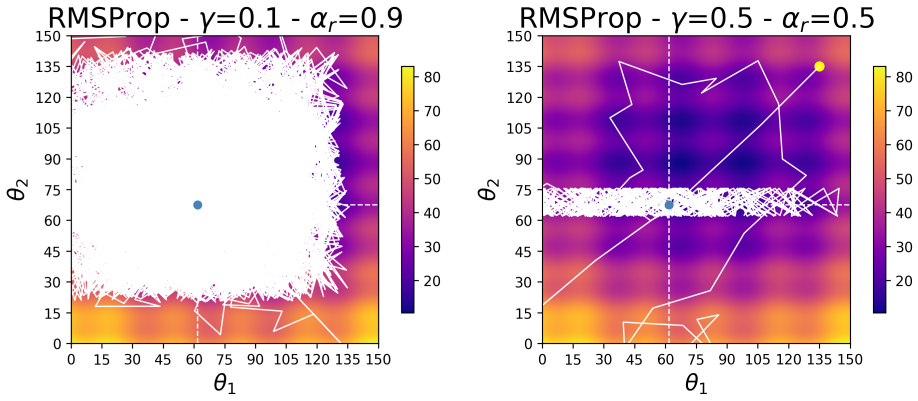


Figure 39: `RMSprop` minimization trajectory for γ=0.1 and α_r=0.9 (a) and for γ=α_r=0.5 (b). The yellow dot represents the starting point $\boldsymbol{\theta}^{(0-1)}$, whereas the blue dot represents the sought minimum.

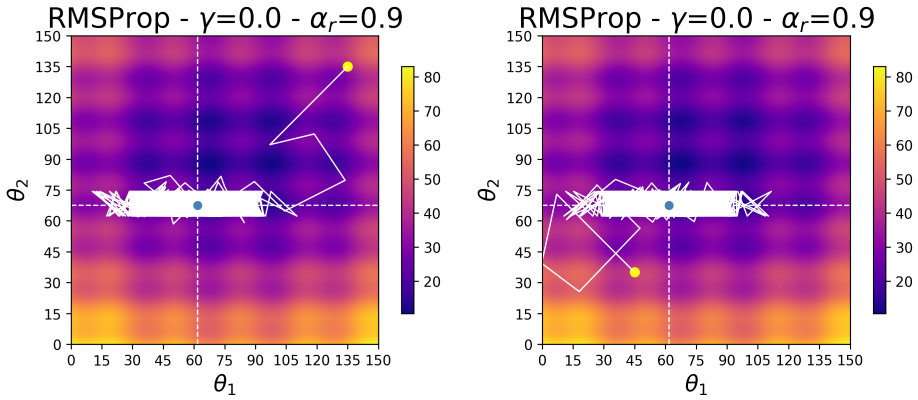


Figure 40: `RMSprop` minimization trajectory for for $\boldsymbol{\theta}^{(0-1)}$ (a) and $\boldsymbol{\theta}^{(0-2)}$ (b). `RMSprop` is featured by γ=0.0 and α_r=0.9 for both cases. The yellow dot represents the starting point $\boldsymbol{\theta}^{(0)}$, whereas the blue dot represents the sought minimum.

use of `AdamW` does not provide an improvement, compared to standard `Adam`, despite the extremely low weight decay penalty equal to 0.00001. `Adam` still represent, for this example, the most robust and efficient optimizer, provided that no tuning is necessary, compared to other optimization techniques.

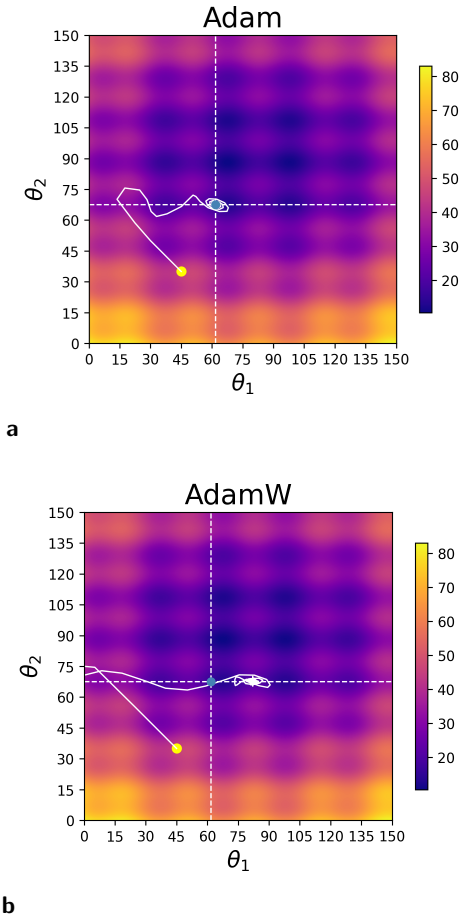


Figure 41: Minimization trajectories for `Adam` (a) and `AdamW` (with weight decay penalty of 0.00001) (b). The yellow dot represents the starting point $\boldsymbol{\theta}^{(0-2)}$, whereas the blue dot represents the sought minimum.

For further insight, see Chapter 2 - *Introduction to regression methods*.

4 Automatic Differentiation

As stated in Section 2, Neural Networks $\mathcal{NN}$ are a special type of statistical models $\mathcal{H}_\theta$, made of complex functions $\boldsymbol{h}_\theta$ defined as composition of non linear ridge activations applied to affine transformations. The problem they aim at solving is stated in problem $(\mathcal{P})$. The search of the "best" set of weights $\boldsymbol{\theta} \in \Theta \subset \mathbb{R}^{d_m}$ is performed by attempting at minimizing the empirical loss

function $L_{\mathcal{D}_{XY}}(\boldsymbol{h}_\theta)$, defined over a set of i.i.d. samples $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, with $\boldsymbol{x}_i \in \mathcal{X} \subset \mathbb{R}^{d_X}$ and $\boldsymbol{y}_i \in \mathcal{Y} \subset \mathbb{R}^{d_Y}$. The quest for the best predictor is a matter of optimization (see Section 3, and the solution $(\mathcal{G})$ of problem $(\mathcal{P})$). Essentially, weights and biases of each neuronal affine transformation are the weights $\boldsymbol{\theta}$ we are looking for. The composition pattern of such weights and biases, within a $\mathcal{NN}$, follows intricate graphs. Therefore, weights and biases are progressively updated via the *backward propagation* algorithm, based on the derivative chain rule.

4.1 Updating weights with the chain rule

If one adopt the standard Stochastic Gradient Descent framework, with mini-baches (see Algorithm 2), the *backward propagation* algorithm consists in computing the derivative of $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_i}$ and the iterative update each weight θ_i according to the following scheme:

$$\begin{aligned}\boldsymbol{\theta}^{(i+1)} =& \boldsymbol{\theta}^{(i)} - \frac{\eta}{N_b} \sum_{k=(j-1)\frac{N_b}{N}}^{j\frac{N_b}{N}} \boldsymbol{\nabla}_\theta l\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}_k\right), \boldsymbol{y}_k\right) \\ & -\eta \cdot \lambda \cdot \sum_{k=(j-1)\frac{N_b}{N}}^{j\frac{N_b}{N}} \boldsymbol{\nabla}_\theta \Omega(\boldsymbol{w}), \quad \eta \in \mathbb{R}^+\end{aligned} \tag{79}$$

with $l\left(\boldsymbol{h}\left(\boldsymbol{x}\right), \boldsymbol{y}\right) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ being a measure of the distance between real label and prediction, such that $l\left(\boldsymbol{y}, \boldsymbol{y}\right) = 0$. $\Omega(\boldsymbol{w})$ represents a penalty on the norm of the weights, with a penalty coefficient λ. The reasons behind weight penalty are outlined in Section 2.1. η is the *learning rate*. The back-propagation is usually performed on mini-batches of N_b instances. Equation (79) represents the so called *batch gradient descent*. The partial derivatives $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_i}$ are computed according to the classical *chain rule*, that reads:

$$\begin{aligned}\boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} &= \frac{1}{N} \sum_{k=1}^{N} \boldsymbol{\nabla}_{\boldsymbol{\theta}} l\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}_k\right), \boldsymbol{y}_k\right) + \lambda \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} \Omega\left(\boldsymbol{\theta}\right) \\ \text{with } \boldsymbol{\nabla}_{\boldsymbol{\theta}} l\left(\boldsymbol{h}_\theta\left(\boldsymbol{x}\right)\right) &= \boldsymbol{\nabla}_\theta \boldsymbol{h}_\theta\left(\boldsymbol{x}\right)^T . \boldsymbol{\nabla}_{h_\theta} l\left(\boldsymbol{h}_\theta\right)\end{aligned} \tag{80}$$

The Jacobian $\mathbb{J}_h^{(k)} = \boldsymbol{\nabla}_\theta \boldsymbol{h}_\theta^{(k)}\left(\boldsymbol{x}\right)$ of the output units at k^{th} layer, with the respect to the weights and biases $\boldsymbol{\theta}^{(k)}$ (of size $u^{(k+1)}$, number of output units). reads:

$$\mathbb{J}_h^{(k)} = \left[\begin{array}{ccc} \frac{\partial \boldsymbol{h}_\theta^{(k)}}{\partial \theta_1^{(k)}} & \frac{\partial \boldsymbol{h}_\theta^{(k)}}{\partial \theta_2^{(k)}} & \cdots \quad \frac{\partial \boldsymbol{h}_\theta^{(k)}}{\partial \theta_{u^{(k+1)}}^{(k)}} \end{array} \right] \tag{81}$$

The $\boldsymbol{\nabla}_{h_\theta} l\left(\boldsymbol{h}_\theta\right)$ represents the gradient of the loss function with the respect to the activation of the output units. Therefore, the chain rule with the respect

to $\boldsymbol{\theta}^{(k)}$ can be explicitly stated as follows:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} l\left(\boldsymbol{h}_{\theta}\left(\boldsymbol{x}\right)\right) = \begin{bmatrix} \frac{\partial l}{\partial \theta_1^{(k)}} \\ \frac{\partial l}{\partial \theta_2^{(k)}} \\ \cdots \\ \frac{\partial l}{\partial \theta_{u^{(k+1)}}^{(k)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial h_1^{(k)}}{\partial \theta_1^{(k)}} & \frac{\partial h_2^{(k)}}{\partial \theta_1^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \theta_1^{(k)}} \\ \frac{\partial h_1^{(k)}}{\partial \theta_2^{(k)}} & \frac{\partial h_2^{(k)}}{\partial \theta_2^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \theta_2^{(k)}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial h_1^{(k)}}{\partial \theta_{u^{(k+1)}}^{(k)}} & \frac{\partial h_2^{(k)}}{\partial \theta_{u^{(k+1)}}^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \theta_{u^{(k+1)}}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_1^{(k)}} \\ \frac{\partial l}{\partial h_2^{(k)}} \\ \cdots \\ \frac{\partial l}{\partial h_{u^{(k+1)}}^{(k)}} \end{bmatrix} \tag{82}$$

The chain "rules" the automatic differentiation In order to better understand the back propagation and automatic differentiation, the following paragraph outlines all the necessary operations to compute the back-propagation for a classification problem with N_y classes, a dataset $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ with $y_i \in \{1, ..., N_y\}$ (see Chapter 4 - *Classification Techniques in Machine Learning*). The $\mathcal{NN}$ classifier is represented by the following $\mathcal{MLP}$ h_θ, defined as:

$$h_{\theta}\left(\boldsymbol{x}\right) = \sigma_y\left(\sum_{c=1}^{N_y} h_c^{(N_\ell)}\left(\boldsymbol{x}\right)\boldsymbol{e}_c\right) \tag{83}$$

where the output activation function $g^{(o)}$ is equal to the softmax$_y$ function, that reads:

$$h^{(o)} = g^{(o)}(\boldsymbol{a}^{(o)}) = \sigma_y\left(\boldsymbol{a}^{(o)}\right) = \text{softmax}_y\left(\boldsymbol{a}^{(o)}\right) = \frac{e^{a_y^{(o)}(\boldsymbol{x})}}{\sum_{c=1}^{N_y} e^{a_c^{(o)}(\boldsymbol{x})}} \tag{84}$$

In particular, the pre-activation function of the output layer reads as N_y entries that read:

$$a_c^{(o)}\left(\boldsymbol{x}\right) = h_c^{(N_\ell)} = g_c^{(N_\ell)}\left(a_c^{(N_\ell)}\left(\boldsymbol{x}\right)\right), \quad 1 \leq c \leq N_y \tag{85}$$

Finally, the classifier is trained to minimize the Negative Log-Likelihood loss that reads:

$$l\left(h_{\theta}\left(\boldsymbol{x}\right), y\right) = -\sum_{c=1}^{N_y} \ln\left(\sigma_c\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)\right)\chi_{(y=c)} \tag{86}$$

The following steps outline the different steps of the back-propagation, applying the chain rule to Equation (86) [10].

[10] The complete tutorial can be found at https://www.youtube.com/watch?v=p5tL2JqCRDo&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=12 (Hugo Larochelle video-lecture series "Neural networks").

1. Derivative at output activation:

$$\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial \sigma_c} = -\frac{\chi_{(y=c)}}{\sigma_c\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)} \tag{87}$$

2. Gradient @ output activation (see softmax_y definition in Equation (91))

$$\begin{aligned}\boldsymbol{\nabla}_\sigma l\left(h_\theta\left(\boldsymbol{x}\right),y\right) &= \sum_{c=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial \sigma_c}\boldsymbol{e}_c = \\ &= -\sum_{c=1}^{N_y}\frac{\chi_{(y=c)}}{\sigma_c\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)}\boldsymbol{e}_c = -\frac{\boldsymbol{e}_y}{\sigma_y\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)}\end{aligned} \tag{88}$$

3. Derivative at activation (output layer N_ℓ, see Equation (85))

$$\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial h_c^{(N_\ell)}} = \sum_{c'=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial \sigma_{c'}}\cdot\frac{\partial \sigma_{c'}}{\partial h_c^{(N_\ell)}}\Big|_{\boldsymbol{a}^{(o)}(\boldsymbol{x})} \tag{89}$$

with:

$$\frac{\partial \sigma_{c'}}{\partial h_c^{(N_\ell)}}\Big|_{\boldsymbol{a}^{(o)}(\boldsymbol{x})} = \frac{\partial \sigma_{c'}}{\partial a_c^{(o)}} = \sigma_{c'}\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)\cdot\left(\chi_{(c'=c)} - \sigma_c\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)\right) \tag{90}$$

4. Gradient at activation (output layer N_ℓ, see Equation (85))
Considering that the softmax output activation function can be rewritten, with the use of the indicator function $\chi_{y=c}$, as:

$$h^{(o)}\left(\boldsymbol{a}^{(o)}\right) = \sigma_y\left(\boldsymbol{a}^{(o)}\right) = \sum_{c=1}^{N_y}\sigma_c\left(\boldsymbol{a}^{(o)}\right)\chi_{y=c} = \left\langle\sum_{c=1}^{N_y}\chi_{c=y}\boldsymbol{e}_c, \boldsymbol{\sigma}\left(\boldsymbol{a}^{(o)}\right)\right\rangle \tag{91}$$

$$\begin{aligned}\boldsymbol{\nabla}_{h^{(N_\ell)}} l\left(h_\theta\left(\boldsymbol{x}\right),y\right) &= \sum_{c=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial h_c^{(N_\ell)}}\boldsymbol{e}_c = \\ &= \sum_{c=1}^{N_y}\sum_{c'=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial \sigma_{c'}}\cdot\frac{\partial \sigma_{c'}}{\partial h_c^{(N_\ell)}}\Big|_{\boldsymbol{a}^{(o)}(\boldsymbol{x})} = \\ &= -\sum_{c=1}^{N_y}\left(\chi_{(y=c)} - \sigma_c\right)\boldsymbol{e}_c = -\left(\boldsymbol{e}_y - \boldsymbol{\sigma}\left(\boldsymbol{a}^{(o)}\right)\left(\boldsymbol{x}\right)\right)\end{aligned} \tag{92}$$

5. Derivative at pre-activation (output layer N_ℓ)

$$\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial a_c^{(N_\ell)}}=\sum_{c'=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial h_{c'}^{(N_\ell)}}\cdot\frac{\partial h_{c'}^{(N_\ell)}}{\partial a_c^{(N_\ell)}} \tag{93}$$

with:

$$\frac{\partial h_{c'}^{(N_\ell)}\left(\boldsymbol{x}\right)}{\partial a_c^{(N_\ell)}}=\frac{\partial g_{c'}^{(N_\ell)}}{\partial a_c^{(N_\ell)}}\left(a_{c'}^{(N_\ell)}\left(\boldsymbol{x}\right)\right)\chi_{(c'=c)} \tag{94}$$

and

$$\begin{aligned}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial a_c}&=-\sum_{c'=1}^{N_y}\left(\chi_{(y=c')}-\sigma_{c'}\right)\frac{\partial g_{c'}}{\partial a_c^{(N_\ell)}}\left(a_{c'}^{(N_\ell)}\left(\boldsymbol{x}\right)\right)\chi_{(c'=c)}=\\&=-\left(\chi_{(y=c)}-\sigma_c\right)\frac{\partial g_c^{(N_\ell)}}{\partial a_c^{(N_\ell)}}\left(a_c^{(N_\ell)}\left(\boldsymbol{x}\right)\right)\end{aligned} \tag{95}$$

6. Gradient at pre-activation (provided that $g_c^{(N_\ell)}=g\quad\forall c$)

$$\begin{aligned}\boldsymbol{\nabla}_{a^{(N_\ell)}}l\left(h_\theta\left(\boldsymbol{x}\right),y\right)&=\sum_{c=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial a_c^{(N_\ell)}}\boldsymbol{e}_c=\\&=\boldsymbol{\nabla}_h l\left(h_\theta\left(\boldsymbol{x}\right),y\right)\odot\boldsymbol{\nabla}_a g=\\&=-\left(\boldsymbol{e}_y-\boldsymbol{\sigma}\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)\right)\odot\boldsymbol{\nabla}_a g\end{aligned} \tag{96}$$

7. Derivative at weight (output layer N_ℓ)
Considering that

$$a_c^{(N_\ell)}=\sum_{c'=1}^{N_y}W_{cc'}^{(N_\ell)}h_{c'}^{(N_{\ell-1})}+b_c^{(N_\ell)} \tag{97}$$

$$\begin{aligned}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial W_{cc'}^{(N_\ell)}}&=\sum_{n=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial a_n^{(N_\ell)}}\frac{\partial a_n^{(N_\ell)}}{\partial W_{cc'}^{(N_\ell)}}=\\&=\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial a_c^{(N_\ell)}}h_{c'}^{(N_\ell-1)}\end{aligned} \tag{98}$$

8. Derivative at bias (output layer N_ℓ)

$$\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial b_c^{(N_\ell)}}=\sum_{n=1}^{N_y}\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial a_n^{(N_\ell)}}\frac{\partial a_n^{(N_\ell)}}{\partial b_c^{(N_\ell)}}=\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right),y\right)}{\partial a_c^{(N_\ell)}}\cdot 1 \tag{99}$$

9. Gradient at weight $W_{cc'}^{(N_\ell)}$ (output layer N_ℓ)
Considering Equation (97) once again:

$$\boldsymbol{\nabla}_{W^{(N_\ell)}} l\left(h_\theta\left(\boldsymbol{x}\right), y\right) = \boldsymbol{\nabla}_a l\left(h_\theta\left(\boldsymbol{x}\right), y\right) \otimes \boldsymbol{h}^{(N_\ell - 1)} \tag{100}$$

10. Gradient at bias (output layer N_ℓ)

$$\boldsymbol{\nabla}_{b^{(N_\ell)}} l\left(h_\theta\left(\boldsymbol{x}\right), y\right) = \boldsymbol{\nabla}_a l\left(h_\theta\left(\boldsymbol{x}\right), y\right) \tag{101}$$

11. Derivative at activation (hidden layer $N_{\ell-1}$)
Considering Equation (97) once again:

$$\frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right), y\right)}{\partial h_c^{(N_{\ell-1})}} = \sum_{n=1}^{N_y} \frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right), y\right)}{\partial a_n^{(N_\ell)}} \frac{\partial a_n^{(N_\ell)}}{\partial h_c^{(N_{\ell-1})}} = \sum_{n=1}^{N_y} \frac{\partial l\left(h_\theta\left(\boldsymbol{x}\right), y\right)}{\partial a_n^{(N_\ell)}} W_{nc}^{(N_\ell)} \tag{102}$$

12. Gradient at activation (hidden layer $N_{\ell-1}$)

$$\boldsymbol{\nabla}_{h^{(N_\ell - 1)}} l\left(h_\theta\left(\boldsymbol{x}\right), y\right) = \boldsymbol{W}^{(N_\ell)T} . \boldsymbol{\nabla}_a l\left(h_\theta\left(\boldsymbol{x}\right), y\right) \tag{103}$$

13. Reiterate points 5 to 13 for $\ell = 1 \ldots N_\ell - 1$, considering that, for all layers $1 \leq k \leq N_\ell$ the following expression holds:

$$h_c^{(k)} = g^{(k)} \left(\sum_{c'=1}^{u^{(k)}} W_{cc'}^{(k)} h^{(k-1)} + b^{(k)} \right) \tag{104}$$

with $u^{(k)}$ being the number of hidden neurons (or units) in layer k

In `PyTorch`, the automatic differentiation is performed by the `autograd` library. `autograd` organizes the different operations in a directed acyclic graph (DAG, see Figure 42). Moreover, it keeps track of each variable (a tensor) along with the executed operations on it (including the gradient of such operations with the respect to the input, stored in the object attribute `grad` and the object function `grad_fn`) and their resulting new tensors. Input tensors are the leaves of the DAG, while the output tensors are the roots of the DAG. Once the `loss` is computed in the `forward` function (a basic method of the class `nn.Module`, that assembles the computational graph), the back-propagation is performed by calling the method `loss.backward()`, that is automatically performed by exploiting the computational graph assembled in the `forward` pass. This is possible because `PyTorch` performs the chain rule operations on the DAG from every operation that involves a gradient-computing tensor or its dependencies, as shown in Figure 43. In order to explicitly update the tensors, one needs to use the context `torch.no_grad()`. The next example clarifies this point.

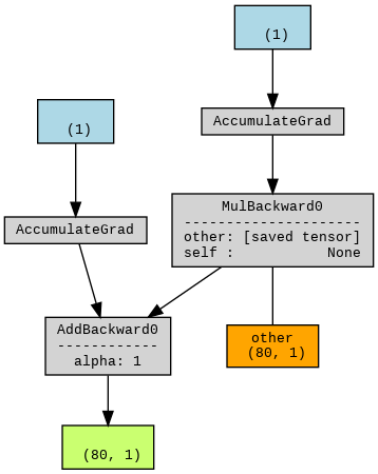


Figure 42: The `PyTorchViz` package allows to visualize the computational graph associated to a `PyTorch` tensor, by adopting the `make_dot` function.

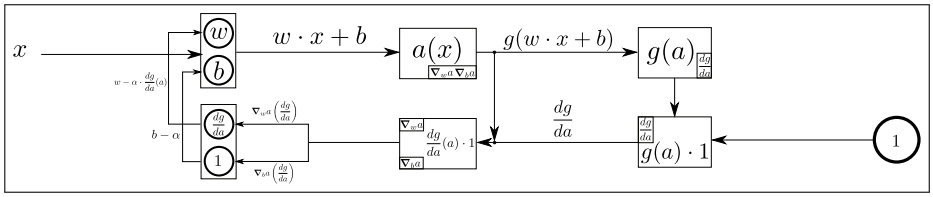


Figure 43: Automatic Differentiation scheme for a function $f(x) = g(a(x))$, with $a = w \cdot x + b$ and g a ridge function. Reprinted and modified from [Fey18]

Example 17. **Automatic Differentiation with `PyTorch`**

```
1
2  # Data for regression problem
3  A = 10.1542550
4  b = 14.7351129
5  epsilon = 0.1
6  N = 200 # number i.i.d. samples
7
8  # Data Generation
9  np.random.seed(42)
10 x = np.random.rand(N, 1)
11 y = b + A * np.exp(x) + epsilon * np.random.randn(N, 1)
12
13 # Shuffles the indices
14 idx = np.arange(N)
15 np.random.shuffle(idx)
16
17 # Uses first 80 random indices for train
18 train_idx = idx[:int(N*0.8)]
19 # Uses the remaining indices for validation
```

```
val_idx = idx[int(N*0.8):]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]

# Convert Numpy arrays into PyTorch Tensors
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

alpha = 1e-1 # learning rate
n_epochs = 1000 # number of epochs

# initialize weights and bias
w = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

for epoch in range(n_epochs):
    yhat = b + w * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    # Perform autodiff!
    loss.backward()

    print(w.grad)
    print(b.grad)

    # We need to use NO_GRAD to manually update the gradients
    with torch.no_grad():
        w -= alpha * w.grad
        b -= alpha * b.grad

    # Drop the old gradients
    w.grad.zero_()
    b.grad.zero_()

print(a, b)
```

In practice, the tensor update is performed automatically too, by using the `PyTorch optimizer` library, presented in Section 3.

Example 18. **Back-propagation through a multi-class classifier with `PyTorch AutoGrad` library**
In this example, a $\mathcal{NN}$ classifier is trained over a dummy dataset (`iris` dataset), with N_y =3 and with two hidden layers of 4 and 8 neurons respectively. The `PyTorchViz` package, one again, allows to visualize the computational graph associated to a `PyTorch` tensor, by adopting the `make_dot` function. In `PyTorch`, compared to Equation (86), the softmax$_y$ vector of probabilities in combination with the Negative Log-Likelihood loss for multi-class classification, can be replaced by creating a model with linear output activation function $g^{(o)}(a_c^{(o)}) = a_c^{(o)}$ and using the `CrossEntropyLoss` class to compute the Negative Log-Likelihood from the so-called `logits` $a_c^{(o)}$, i.e., the *score* obtained by the classifier h_θ for each class. The cross entropy $\mathbb{H}(p\|p_\theta)$, between two probability distributions: $p\left(y_1, y_2, \ldots y_{N_y}\right)$ (the "real" probability distribution

of the classes) and the model probability $p_\theta\left(y|\boldsymbol{x}\right)$:

$$\mathbb{H}\left(p||p_\theta\right) = -\sum_{c=1}^{N_y} p(y_i = 1, y_{j\neq i} = 0) \cdot \ln p_\theta\left(y_i|\boldsymbol{x}\right) \tag{105}$$

which corresponds to Equation (86).

```
 # Source:
 # https://machinelearningmastery.com/building-a-multiclass-classification-model-in-pytorch/
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import copy
# uncomment the next line to install torchviz if necessary
# ! pip install torchviz
from torchviz import make_dot

# To assure reproductibility, fix the random seed
torch.manual_seed(0) # to assure reproductibility
np.random.seed(0) # to assure reproductibility

# Download and parse the IRIS dataset (from the UCI Machine Learning repository)
# This dataset was conceived by Sir Ronald Fisher and it is among the best-known
# dataset for pattern recognition
data_url= "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
data = pd.read_csv(data_url, header=None)
X = data.iloc[:, 0:4]
y = data.iloc[:, 4:]

# The IRIS dataset is composed of the three class labels:
# 1. Iris-setosa
# 2. Iris-versicolor
# 3. Iris-virginica
# apply one-hot encoding
ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False).fit(y)
y = ohe.transform(y)

# convert pandas DataFrame (X) and numpy array (y) into PyTorch tensors
X = torch.tensor(X.values, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

# split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)

# define the MLP sequentially with torch.nn
h_theta = nn.Sequential()
# first hidden layer pre-activation a^(1)
h_theta.add_module('a1', nn.Linear(4, 8))
# first hidden layer activation function g^(1)
h_theta.add_module('g1', nn.ReLU())
# second hidden layer pre-activation a^2
h_theta.add_module('a2', nn.Linear(8, 3))

# define the loss function
loss_fn = nn.CrossEntropyLoss() # loss function
optimizer = optim.Adam(h_theta.parameters(), lr=0.001) # optimizer

```

```python
# prepare model and training parameters
n_epochs = 10 # number of training epochs
batch_size = 5 # size of the mini batch
batches_per_epoch = len(X_train) // batch_size

best_acc = - np.inf  # init to negative infinity
best_weights = None
# track loss history
train_loss_hist = []
train_acc_hist = []
test_loss_hist = []
test_acc_hist = []

# training loop over the epochs
for epoch in range(n_epochs):
    epoch_loss = []
    epoch_acc = []
    # set model in training mode
    h_theta.train()
    with tqdm.trange(batches_per_epoch, unit="batch", mininterval=0) as bar:
        bar.set_description(f"Epoch {epoch}")
        for i in bar:
            # take a batch
            start = i * batch_size
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            # infer (forward)
            y_pred = h_theta(X_batch)
            # compute the loss
            loss = loss_fn(y_pred, y_batch)
            # reset previously saved gradients and empty the optimizer memory
            optimizer.zero_grad()
            # run backward propagation
            loss.backward()
            # update weights
            optimizer.step()
            # compute and store metrics
            acc = (torch.argmax(y_pred, 1) == torch.argmax(y_batch, 1)).float().mean()
            epoch_loss.append(float(loss))
            epoch_acc.append(float(acc))
            bar.set_postfix(
                loss=float(loss),
                acc=float(acc)
            )
    # set model in evaluation mode to infer the class in the test set
    # without storing gradients for brackprop
    h_theta.eval()
    # infer the class over the test set
    y_pred = h_theta(X_test)
    ce = loss_fn(y_pred, y_test)


    acc = (torch.argmax(y_pred, 1) == torch.argmax(y_test, 1)).float().mean()
    ce = float(ce)
    acc = float(acc)
    train_loss_hist.append(np.mean(epoch_loss))
    train_acc_hist.append(np.mean(epoch_acc))
    test_loss_hist.append(ce)
    test_acc_hist.append(acc)
    if acc > best_acc:
        best_acc = acc
        best_weights = copy.deepcopy(h_theta.state_dict())
    print(f"Epoch {epoch} validation: Cross-entropy={ce:.2f}, Accuracy={acc*100:.1f}%")

# plot computational graph with torchviz
h_theta.eval()
```

```
y_pred = h_theta(X_test)
ce = loss_fn(y_pred, y_test)
make_dot(y_pred, params=dict(h_theta.named_parameters()),
 show_attrs=True, show_saved=True).render("classifier_graph", format="png")
make_dot(ce, params=dict(h_theta.named_parameters()),
 show_attrs=True, show_saved=True).render("loss_graph", format="png")



# Restore best model
h_theta.load_state_dict(best_weights)

# Plot the loss and accuracy for train and test sets
plt.plot(train_loss_hist, label="train")
plt.plot(test_loss_hist, label="test")
plt.xlabel("epochs")
plt.ylabel("cross entropy")
plt.legend()
plt.show()

plt.plot(train_acc_hist, label="train")
plt.plot(test_acc_hist, label="test")
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.legend()
plt.show()
```

4.2 Countermeasures to vanishing gradients

Automatic Differentiation is based on the classical *chain rule* outlined in Equation (80). During backpropagation, gradients that "flow" into a neuron are proportional to the activation of the input neurons they are connected with. When the activation values are small, gradients vanish and the neurons do not learn anything. If one considers a feed-forward Multi-Layer Perceptron on the dataset $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$, with $\boldsymbol{x}_i \in \mathbb{R}^{d_X}$, $y \in \mathbb{R}$, made of N_ℓ hidden layers reads:

$$\begin{aligned} h_\theta(\boldsymbol{x}) =& g^{(o)} \circ a^{(o)} \circ g^{(N_\ell)} \circ a^{(N_\ell)} \circ \ldots \circ g^{(\ell)} \circ a^{(\ell)} \circ \ldots g^{(1)} \circ a^{(1)}(\boldsymbol{x}) \\ a^{(\ell)}(u) =& w^{(\ell)} \cdot u + b^{(\ell)}, \quad 1 < \ell \leq N_\ell \\ h^{(\ell)}(u) =& g^{(\ell)}\left(a^{(\ell)}(u)\right), \quad 1 \leq \ell \leq N_\ell, \quad h^{(o)}(u) = g^{(o)}(u) \\ a^{(1)}(\boldsymbol{x}) =& \left\langle \boldsymbol{w}^{(1)}, \boldsymbol{x} \right\rangle + b^{(1)}, \end{aligned} \tag{106}$$

If one computes the derivative of the loss function $L_{\mathcal{D}_{XY}}$ is based on the chain rule. For instance, in order to update $w_c^{(1)}$ the following derivative must be computed:

$$\frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h^{(N_\ell)}} \left(\prod_{\substack{\ell=1 \\ N_\ell > 1}}^{N_\ell - 1} \frac{\partial g^{(N_\ell + 1 - \ell)}}{\partial a^{(N_\ell + 1 - \ell)}} \cdot \frac{\partial a^{(N_\ell + 1 - \ell)}}{\partial h^{(N_\ell - \ell)}} \right) \frac{\partial g^{(1)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial w_c^{(1)}} \tag{107}$$

with the following update rule:

$$w_c^{(1)(i+1)} = w_c^{(1)(i)} - \eta \frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}} \tag{108}$$

A vanishing gradient make the iterative back-propagation scheme to fail, since the weights tends to remain constant due to the fact that $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}} \approx 0$. The choice of the activation functions g plays a major role in making the gradient vanish. In particular, the activation functions g with limited *output range*, i.e., whose derivative $\frac{\partial g}{\partial a}$ has a compact support (the so called *region of significance*), can yield derivatives close to 0 for certain values of the pre-activation a (see Figure 44).

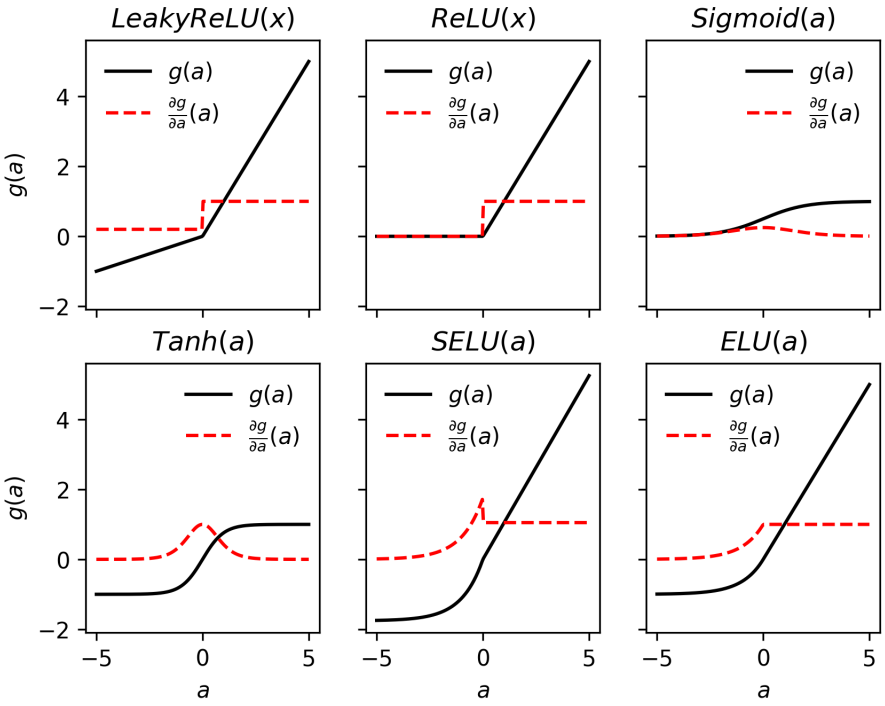


Figure 44: Examples of activations functions $g(a)$ and their derivative with the respect to the pre-activation function a.

The gradient $\boldsymbol{\nabla}_{w^{(1)}} L_{\mathcal{D}_{XY}}$ tends to vanish when the derivative

$$\frac{\partial g^{(N_\ell+1-\ell)}}{\partial a^{(N_\ell+1-\ell)}} \approx 0 \tag{109}$$

This effect is even emphasized for deeper networks, because the product

$$\prod_{\ell=1}^{N_\ell-1} \frac{\partial g^{(N_\ell+1-\ell)}}{\partial a^{(N_\ell+1-\ell)}} \cdot \frac{\partial a^{(N_\ell+1-\ell)}}{\partial h^{(N_\ell-\ell)}} \approx 0 \tag{110}$$

even faster if the same activation function g is chosen for all layers.
The most commonly adopted activation function has been $g = ReLU$ so far [GBB11]. As a matter of fact, deep $\mathcal{NN}$ featured by $ReLU$ with hard zero threshold, in combination with ℓ^1-norms on the weights (to bound the positive activation values $a > 0$), promote sparsity and ease the gradient back-propagation on active units [GBB11]. After being initialized with uniform sampling, almost half of the hidden neurons output value is zero, with this fraction increasing whenever the ℓ^1 regularization is adopted. In this sense, active neurons alleviates the vanishing gradient since $\frac{\partial g}{\partial a}|_{a>0} = 1$. The $\mathcal{MLP}$ becomes a piece-wise linear function, with neurons that are either deactivated $a < 0$ or operating in linear regime $a > 0$. The basis of those sparse piece-wise linear ridge functions helps representing the labelling of regression function $f : \boldsymbol{x} \mapsto y$ over a basis of ridge functions (see Section 2.1 for further details about the Universal Approximation Theorem).

Example 19. **Vanishing gradient effect for deep $\mathcal{MLP}$ classifiers**
In this example, a 10-layer deep $\mathcal{MLP}$, is conceived for the multi-class classification problem outlined in Section 4.1, with N_y=3 classes and with $fan_{in} = fan_{out}$ =5 for all hidden layers (see Example 18 for further details). The depth of the $\mathcal{MLP}$ designed in the present `PyTorch` example makes it prone to vanishing gradients, at the deepest layers especially. In particular, the example compares the evolution of the mean (over the database samples) gradient norm with the training epochs, for two different activation functions, namely $tanh$ and $ReLU$. The results are depicted in Figure 45. From Figure 45, it is clear that :

- When the activation function saturates and the $\mathcal{MLP}$ has too many layers, the gradient of the loss function with the respect to the weights and biases of the initial layers vanishes, preventing the loss to converge to 0 (see Figure 45a and Figure 45c).
- Replacing the activation function by a non-saturating one, such as $g^{(\ell)} = ReLU$, prevents the gradient to vanish at all layers which in turns make the loss function converge to 0.

The `PyTorch` code is reported below.

```
# Source:
# https://machinelearningmastery.com/building-a-multiclass-classification-model-in-pytorch/
# https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
```

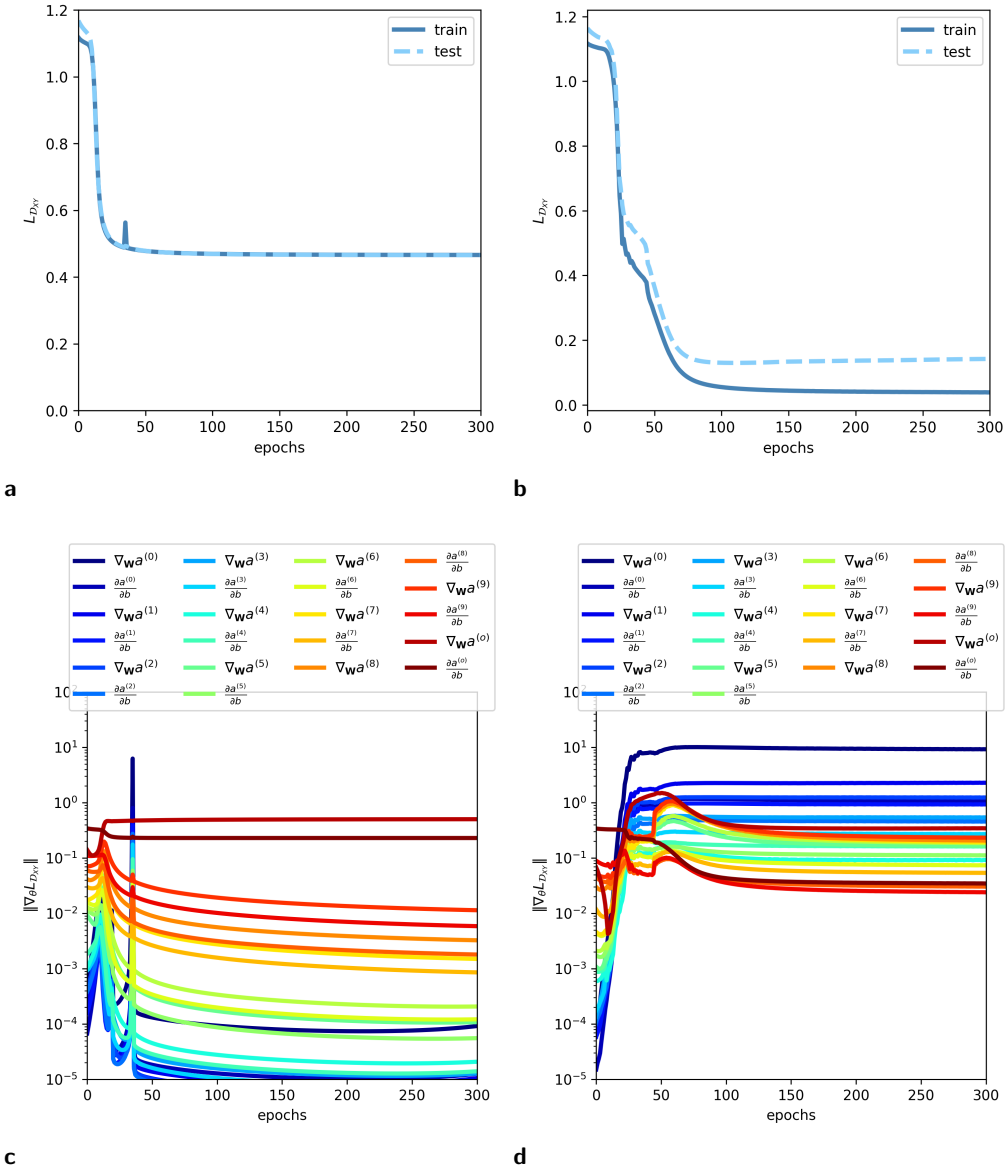


Figure 45: Loss function evolution with epoch of training (a-b) and mean gradient norm of each layer of the $\mathcal{MLP}$ (c-d). (a) and (c) refer to the case with the activation function $g^{(\ell)} = tanh$, whereas (b) and (d) refer to the case with activation function $g^{(\ell)} = ReLU$.

```
import torch.nn as nn
import torch.optim as optim
import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import copy
# install torchviz (optional)
# ! pip install torchviz
# from torchviz import make_dot

# To assure reproductibility, fix the random seed
torch.manual_seed(0) # to assure reproductibility
np.random.seed(0) # to assure reproductibility

# Download and parse the IRIS dataset (from the UCI Machine Learning repository)
# This dataset was conceived by Sir Ronald Fisher and it is among the best-known
# dataset for pattern recognition
data_url= "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
data = pd.read_csv(data_url, header=None)
X = data.iloc[:, 0:4]
y = data.iloc[:, 4:]

# The IRIS dataset is composed of the three class labels:
# 1. Iris-setosa
# 2. Iris-versicolor
# 3. Iris-virginica
# apply one-hot encoding
ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False).fit(y)
y = ohe.transform(y)

# convert pandas DataFrame (X) and numpy array (y) into PyTorch tensors
X = torch.tensor(X.values, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

# split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)

# Function to track grad norm
def track_grad_norm(model, norm_type='fro'):
  # Collect parameters (weights and biases from all layers)
  parameters = [p for p in model.parameters()
                if p.grad is not None and p.requires_grad]
  # Defin the total norm and initialize to 0
  total_norm = {k: 0.0 for k,_ in model.named_parameters()}
  if len(parameters) > 0:
      # get weight device (initial weight)
      device = parameters[0].grad.device
      # Assign values of the grad to each layer (in the dictionary)
      for n,p in model.named_parameters():
          total_norm[n]=torch.norm(p.grad.detach(), norm_type).to(device).item()
  return total_norm

loss_fn = nn.CrossEntropyLoss() # loss function

# Design the MLP
h_theta = nn.Sequential()
nl = 10 # number of hidden layers
fan_in = 5 # number of hidden units per neuron
# Define the dictionary of different activation functions
activations={'tanh': nn.tanh(), 'ReLU': nn.ReLU(), 'Sigmoid': nn.Sigmoid() }

# chosen activation function
act = 'tanh' # switch to ReLU or Sigmoid
activation = activations[act]
# Initialize latex dictionary for plotting
latex_dict={}
```

```
# Iterate over layer number
for l in range(nl):
    if l==0:
        h_theta.add_module('a{:>d}'.format(l), nn.Linear(d_X, fan_in))
    else:
        h_theta.add_module('a{:>d}'.format(l), nn.Linear(fan_in, fan_in))

    h_theta.add_module('g{:>d}'.format(l), activation)
    # Update the latex dictionary for plotting
    latex_dict["a{:>d}.weight".format(l)]=r"$\nabla_{{ \mathbf{{W}} }} a^{{({:>d})}}$".format(l)
    latex_dict["a{:>d}.bias".format(l)]=r"$\frac{{\partial a^{{({:>d})}}}}{{\partial b}}$".format(l)
# define output layer
h_theta.add_module('aout', nn.Linear(fan_in, Ny))
# Update the latex dictionary for plotting
latex_dict['aout.weight'.format(l)]=r"$\nabla_{{ \mathbf{{W}} }} a^{{(o)}}$"
latex_dict['aout.bias'.format(l)]=r"$\frac{{\partial a^{{(o)}}}}{{\partial b}}$"

# Initialize weights
# h_theta.apply(kaiming_normal_init)
init_weight=False

optimizer = optim.Adam(h_theta.parameters(), lr=0.001) # optimizer


# prepare model and training parameters
n_epochs = 300 # number of training epochs
batch_size = 5 # size of the mini batch
batches_per_epoch = len(X_train) // batch_size

best_acc = - np.inf    # init to negative infinity
best_weights = None
# track loss history
train_loss_hist = []
train_acc_hist = []
test_loss_hist = []
test_acc_hist = []
norm_grad={n:[] for n,_ in h_theta.named_parameters()}


# # Initialize weights
# h_theta.apply(xavier_normal_init)
# training loop over the epochs
for epoch in range(n_epochs):
    epoch_loss = []
    epoch_acc = []
    epoch_norm_grad = {n:[] for n,_ in h_theta.named_parameters()}
    # set model in training mode
    h_theta.train()

    with tqdm.trange(batches_per_epoch, unit="batch", mininterval=0) as bar:
        bar.set_description(f"Epoch {epoch}")
        for i in bar:
            # take a batch
            start = i * batch_size
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            # infer (forward)
            y_pred = h_theta(X_batch)
            # compute the loss
            loss = loss_fn(y_pred, y_batch)
            # reset previously saved gradients and empty the optimizer memory
            optimizer.zero_grad()
            # run backward propagation
            loss.backward()
            # update weights
            optimizer.step()
```

```
            # compute and store metrics
            acc = (torch.argmax(y_pred, 1) == torch.argmax(y_batch, 1)).float().mean()
            epoch_loss.append(float(loss))
            epoch_acc.append(float(acc))
            bar.set_postfix(
                loss=float(loss),
                acc=float(acc)
            )
            # Update the norm of the grad per sample
            update_grad_norm_batch = track_grad_norm(h_theta)
            for p,ng in update_grad_norm_batch.items():
              epoch_norm_grad[p].append(ng)
    # Compute the mean norm of the grad
    for p,ng in epoch_norm_grad.items():
      norm_grad[p].append(np.mean(ng))
    # set model in evaluation mode to infer the class in the test set
    # without storing gradients for brackprop
    h_theta.eval()
    # infer the class over the test set
    y_pred = h_theta(X_test)
    ce = loss_fn(y_pred, y_test)

    acc = (torch.argmax(y_pred, 1) == torch.argmax(y_test, 1)).float().mean()
    ce = float(ce)
    acc = float(acc)
    train_loss_hist.append(np.mean(epoch_loss))
    train_acc_hist.append(np.mean(epoch_acc))
    test_loss_hist.append(ce)
    test_acc_hist.append(acc)
    # norm_grad.append(np.mean(epoch_norm_grad))
    # if acc > best_acc:
    #     best_acc = acc
    #     best_weights = copy.deepcopy(h_theta.state_dict())
    print(f"Epoch {epoch} validation: Cross-entropy={ce:.2f}, Accuracy={acc*100:.1f}%")

# Plot the loss for train and test sets
fig, ax = plt.subplots(figsize=(5,5))
ax.plot(train_loss_hist, label="train",color='steelblue',linewidth=3)
ax.plot(test_loss_hist, label="test",color='lightskyblue',linewidth=3,linestyle='--')
ax.set_xlabel(r"epochs")
ax.set_ylabel(r"$\mathit{L}_{\mathcal{D}_{XY}}$")
ax.set_xlim(0,300)
ax.set_ylim(0.0,1.2)
ax.legend()

if init_weight:
  fig.savefig("loss_classifier_init_{:>d}_{:>d}_{:>s}.png".format(nl, fan_in, act),
              dpi=300, bbox_inches="tight")
else:
  fig.savefig("loss_classifier_{:>d}_{:>d}_{:>s}.png".format(nl, fan_in, act),
              dpi=300, bbox_inches="tight")

# Plot the mean gradient norm
colors = [plt.cm.jet(c) for c in  np.linspace(0,1,len(list(norm_grad.keys())))  ]

fig, ax = plt.subplots(figsize=(5,5))
for i, (n, ng) in enumerate(norm_grad.items()):
  ax.semilogy(ng, label=latex_dict[n],color=colors[i],linewidth=3)
# ax.plot(test_loss_hist, label="test")
ax.set_xlabel("epochs")
ax.set_ylabel(r"$\Vert \nabla_\theta \mathit{L}_{\mathcal{D}_{XY}}\Vert$")
ax.set_xlim(0,300)
ax.set_ylim(1e-5,1e2)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.4),ncol=4,)
if init_weight:
  fig.savefig("norm_grad_classifier_init_{:>d}_{:>d}_{:>s}.png".format(nl, fan_in, act),
```

```
                dpi=300, bbox_inches="tight")
else:
    fig.savefig("norm_grad_classifier_{:>d}_{:>d}_{:>s}.png".format(nl, fan_in, act),
                dpi=300, bbox_inches="tight")

```

4.2.1 Weight initialization

The weight initialization can play a major role case of vanishing gradient, since, depending on the activation function at stake, for deeper neural networks, it can occur that $\boldsymbol{\nabla}_{w^{(1)}} L_{\mathcal{D}_{XY}} \approx 0$ in Equation (107), whenever the following condition occurs:

$$\frac{\partial a^{(N_\ell - \ell)}}{\partial h^{(N_\ell - \ell - 1)}} = w^{(N_\ell - \ell)} \approx 0 \tag{111}$$

Weight initialization was firstly introduced in the seminal work of Bengio et al. [Ben+06]. The authors proposed to optimize the weights of a $\mathcal{NN}$ by greedy unsupervised steps, i.e., by training one layer at time and freezing all the previous ones. This approach, by late 2000s, was the leading strategy to avoid trivial initializations, such as zero or constant initialization. The latter does not necessarily imply that the gradient vanishes, provided that the network is shallow though, as for instance in case of a $\mathcal{MLP}$ $N_\ell = 1$. For the latter, Equation (107) simplifies in the following expression:

$$\frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h^{(1)}} \frac{\partial g^{(1)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial w_c^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial a^{(1)}} \cdot x_c \tag{112}$$

If two neurons are added to the $\mathcal{MLP}$ with one hidden layer, Equation (112) becomes:

$$\begin{cases} \dfrac{\partial L_{\mathcal{D}_{XY}}}{\partial w_{1c}^{(1)}} = \dfrac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \dfrac{\partial a^{(o)}}{\partial h_1^{(1)}} \dfrac{\partial h_1^{(1)}}{\partial a_1^{(1)}} \cdot x_c & (113) \\ \dfrac{\partial L_{\mathcal{D}_{XY}}}{\partial w_{2c}^{(1)}} = \dfrac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \dfrac{\partial a^{(o)}}{\partial h_2^{(1)}} \dfrac{\partial h_2^{(1)}}{\partial a_2^{(1)}} \cdot x_c & (114) \end{cases}$$

with $\boldsymbol{a}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$, and $h_i^{(1)} = g(a_i^{(1)})$ (see Figure 1). In this case, the trivial initialization $\boldsymbol{W} = cst.$ and $\boldsymbol{b}^{(1)} = cst.$ does not make the gradient vanish (if and only if $N_\ell = 1$) but since $a_1^{(1)} = a_2^{(1)} = 0$, then $h_1^{(1)} = g(a_1^{(1)}) = h_1^{(1)} = g(a_1^{(1)})$, so that $w_{1c}^{(1)(i)} = w_{2c}^{(1)(i)}$ until convergence. This symmetric evolution of the weights is rather restrictive though, since it cannot be prevented after the first gradient descent iteration.

A smarter weight initialization strategy is to sample them from the standard normal or uniform distribution. However, this strategy can incur into either vanishing gradients, because the activation function can saturate for large random samples (see Equation (109)) or because the weight value is too small (see Equation (111)). When randomly sampled, the weight value is dominated by

the choice of their variance. In other words, the optimum sampling solution would be an unbiased random sampling with the "optimum" variance that avoid vanishing gradients. The zero-mean avoids introducing spurious bias shifts (see Section 5.2 and [CUH16]) and asymmetric weight configurations. The "optimum" variance should be chosen in order to decorrelate weights, biases and input in the pre-activation output $a_i^{(k)}$ at each layer k (see Equation (97)), so that its variance reads:

$$\mathbb{V}(a_i^{(k)}) = \mathbb{V}\left(\sum_{j=1}^{u^{(k)}} W_{ij}^{(k)} h_j^{(k-1)}\right) + \mathbb{V}\left(b_i^{(k)}\right) \tag{115}$$

with

$$\begin{aligned}\mathbb{V}\left(\sum_{j=1}^{u^{(k)}} W_{ij}^{(k)} h_j^{(k-1)}\right) = &\sum_{j=1}^{u^{(k)}} \mathbb{V}\left(W_{ij}^{(k)}\right) \cdot \left(\mathbb{E}\left[h_j^{(k-1)}\right]\right)^2 + \\ &+ \sum_{j=1}^{u^{(k)}} \left(\mathbb{E}\left[W_{ij}^{(k)}\right]\right)^2 \cdot \mathbb{V}\left(h_j^{(k-1)}\right) + \\ &+ \sum_{j=1}^{u^{(k)}} \mathbb{V}\left(W_{ij}^{(k)}\right) \cdot \mathbb{V}\left(h_j^{(k-1)}\right)\end{aligned} \tag{116}$$

If one assumes that unbiased weights, biases and inputs, with constant variances $\mathbb{V}(W^{(k)})$, $\mathbb{V}(b^{(k)})$ and $\mathbb{V}(h^{(k-1)})$ respectively, Equation (116) can be rewritten as:

$$\mathbb{V}\left(a_i^{(k)}\right) = \left(u^{(k)} \cdot \mathbb{V}(W^{(k)})\right)^i \cdot \mathbb{V}(h^{(k-1)}) + \mathbb{V}\left(b_i^{(k)}\right) \tag{117}$$

In order to assure that weights, biases and inputs are decorrelated, Equation (117) must hold, by avoiding the quantity $\left(u^{(k)} \cdot \mathbb{V}(W^{(k)})\right)^i$ to vanish or explode. In order to do so, the most effective strategy consists into initializing the biases to 0 and the weight variance as follows:

$$\mathbb{V}(W^{(k)}) = \frac{1}{u^{(k)}} \Rightarrow \mathbb{V}\left(a_i^{(k)}\right) = \mathbb{V}(h^{(k-1)}) + \mathbb{V}\left(b_i^{(k)}\right) \tag{118}$$

or, in order to account for both forward and backward propagation, the Xavier (or Glorot) initialization proposed by [GB10] is preferred:

$$\mathbb{V}(W^{(k)}) = \frac{2}{u^{(k)} + u^{(k+1)}} \tag{119}$$

Xavier (or Glorot) initialization in Equation (119) avoids the pre-activation variance to explode or to vanish both in forward and in backward propagation, with opposite flow of information. This initialization grants a good stability of the training scheme. If the weights are sampled with standard normal distribution, Equation (119) provides the standard deviation. If a uniform distribution

$\mathcal{U}(-c, c)$ is chosen, its variance being $\frac{c^2}{3}$, then the end points can be selected as $\pm\sqrt{\frac{6}{u^{(k)}+u^{(k+1)}}}$.
However, [Kum17] proved that Xavier (or Glorot) initialization seems to highly impact the backward propagation of $\mathcal{NN}$ featured by $ReLU$ activation functions, due to the asymmetric nature of this activation function that is identically zero for negative argument. Therefore, [He+15] proposed to initialize the weight variance as follows:

$$\mathbb{V}(W^{(k)}) = \frac{2}{u^{(k)}} \tag{120}$$

with end point for uniform distribution $\pm\sqrt{\frac{6}{u^{(k)}}}$. This strategy goes under the name of He (or Kaiming) initialization. For both Xavier (or Glorot) and He (or Kaiming) initialization, a gain can be multiplied to the variance initialization.

Example 20. Different weight initialization techniques for a multi-class classifier with `PyTorch`.
In the literature, the number of input units $u^{(k)}$ and the number of output units $u^{(k+1)}$ are referred as to fan_{in} and fan_{out} respectively. `PyTorch` class `torch.nn.init`[11], adopted to perform weight initialization, follows the same notation. The following example shows the weight distribution for different initializations, for a 1-layer $\mathcal{MLP}$ with fan_{in} =100 and fan_{out}=100.

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
# import pandas as pd
import torch
import torch.nn as nn

# define the 1-layer MLP
h_theta = nn.Sequential()
h_theta.add_module('a1', nn.Linear(100, 100))
h_theta.add_module('g1', nn.ReLU())

# define initialization functions
def constant_init(m):
    if type(m) == nn.Linear:
        torch.nn.init.constant_(m.weight, 2)

def uniform_init(m):
    if type(m) == nn.Linear:
        torch.nn.init.uniform_(m.weight,a=-0.5, b=0.5)

def normal_init(m):
    if type(m) == nn.Linear:
        torch.nn.init.normal_(m.weight)

def xavier_uniform_init(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform_(m.weight)

def xavier_normal_init(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_normal_(m.weight)
```

[11] https://pytorch.org/docs/stable/nn.init.html

```python

def kaiming_uniform_init(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_uniform_(m.weight)

def kaiming_normal_init(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_normal_(m.weight)


# Applying different initializations to model

# constant
h_theta.apply(constant_init)
constant_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
constant_weights = torch.concatenate(constant_weights).detach().cpu().numpy()

# uniform
h_theta.apply(uniform_init)
uniform_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
uniform_weights = torch.concatenate(uniform_weights).detach().cpu().numpy()

# normal
h_theta.apply(normal_init)
normal_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
normal_weights = torch.concatenate(normal_weights).detach().cpu().numpy()

# xavier uniform
h_theta.apply(xavier_uniform_init)
xavier_uniform_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
xavier_uniform_weights = torch.concatenate(xavier_uniform_weights).detach().cpu().numpy()

# xavier normal
h_theta.apply(xavier_normal_init)
xavier_normal_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
xavier_normal_weights = torch.concatenate(xavier_normal_weights).detach().cpu().numpy()

# kaiming uniform
h_theta.apply(kaiming_uniform_init)
kaiming_uniform_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
kaiming_uniform_weights = torch.concatenate(kaiming_uniform_weights).detach().cpu().numpy()

# kaiming normal
h_theta.apply(kaiming_normal_init)
kaiming_normal_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
kaiming_normal_weights = torch.concatenate(kaiming_normal_weights).detach().cpu().numpy()

weights = {"uniform": uniform_weights,
           "normal": normal_weights,
           "Xavier uniform": xavier_uniform_weights,
           "Xavier normal": xavier_normal_weights,
           "Kaiming uniform": kaiming_uniform_weights,
           "Kaiming normal": kaiming_normal_weights}

# plot histograms
fig, ax = plt.subplots(nrows=2, ncols=3, sharex=True, sharey=True)

ax=ax.flatten()

for a, (k,v) in enumerate(weights.items()):
  # compute histogram
  counts, bins = np.histogram(v, bins=v.size//10)
  ax[a].hist(bins[:-1], bins, weights=counts,
      align='left', color='k', density=True,
      log=True)
  ax[a].set_title(k)
```

```
    if a>2:
        ax[a].set_xlabel(r"$W_{ij}$")
    if a==0 or a==3:
        ax[a].set_ylabel("#")
    ax[a].set_xlim(-3.0,3.0)
    ax[a].set_ylim(0,5.0)
fig.savefig("weight_initialization.png",dpi=300,bbox_inches='tight')
```

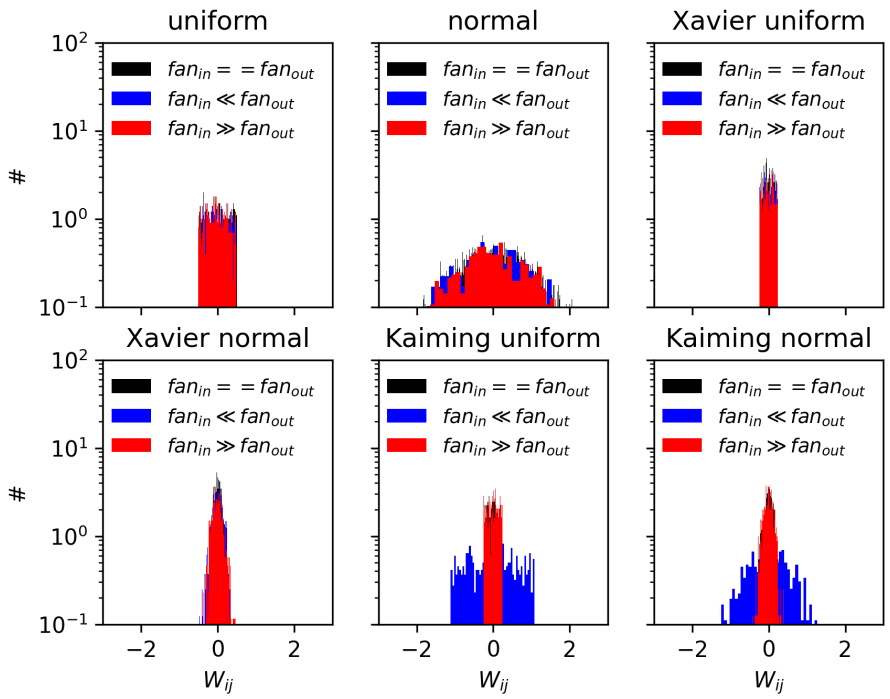


Figure 46

As shown in Figure 46, the weight distribution for Kaiming is highly affected by the value of fan_{in}, whereas the other distributions remain stable, since they either do depend on both fan_{in}

References

[ACB17] **Arjovsky, Martin, Soumith Chintala, and Léon Bottou**. "Wasserstein generative adversarial networks". In: *International conference on machine learning*. PMLR, 2017, pp. 214–223.

[Bac17] **Bach, Francis**. "Breaking the curse of dimensionality with convex neural networks". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 629–681.

[Bar93] **Barron, Andrew R**. "Universal approximation bounds for superpositions of a sigmoidal function". In: *IEEE Transactions on Information theory* 39.3 (1993), pp. 930–945.

[Ben+06] **Bengio, Yoshua et al.** "Greedy layer-wise training of deep networks". In: *Advances in neural information processing systems* 19 (2006).

[BSF94] **Bengio, Yoshua, Patrice Simard, and Paolo Frasconi**. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[Cam19] **Campagne, J.E.** *L'apprentissage par réseaux de neurones profonds*. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2019). 2019. DOI: https://doi.org/10.4000/annuaire-cdf.16767. URL: https://www.di.ens.fr/~mallat/College/Cours-2019-Mallat-Jean-Eric-Campage.pdf.

[Cam20] **Campagne, J.E.** *Modè les multi-échelles et réseaux de neurones convolutifs*. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2020). 2020. URL: https://www.di.ens.fr/~mallat/College/Cours2020-Mallat-Jean-Eric-Campagne.pdf.

[CBB97] **Cun, Yann Le, L. Bottou, and Y. Bengio**. "Reading checks with multilayer graph transformer networks". In: *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 1. 1997, 151–154 vol.1. DOI: 10.1109/ICASSP.1997.599580.

[CGL21] **Castro-Cruz, David, Filippo Gatti, and Fernando Lopez-Caballero**. "Assessing the impact of regional geology on the ground motion model variability at the Kashiwazaki-Kariwa Nuclear Power Plant (Japan) via physics-based numerical simulation". en. In: *Soil Dynamics and Earthquake Engineering* 150 (Nov. 2021), p. 106947. ISSN: 02677261. DOI: 10.1016/j.soildyn.2021.106947. URL: https://linkinghub.elsevier.com/retrieve/pii/S0267726121003699.

[Che+14] **Chen, Liang-Chieh et al.** "Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs". en. In: arXiv:1412.7062 (June 2014). arXiv:1412.7062 [cs]. URL: http://arxiv.org/abs/1412.7062.

[CUH16] **Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter**. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". en. In: arXiv:1511.07289 (Feb. 2016). arXiv:1511.07289 [cs]. URL: http://arxiv.org/abs/1511.07289.

[DHS11] **Duchi, John, Elad Hazan, and Yoram Singer**. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimiza-

tion". In: *J. Mach. Learn. Res.* 12.null (July 2011), pp. 2121–2159. ISSN: 1532-4435.

[DV18] **Dumoulin, Vincent and Francesco Visin**. "A guide to convolution arithmetic for deep learning". en. In: arXiv:1603.07285 (Jan. 2018). arXiv:1603.07285 [cs, stat]. URL: http://arxiv.org/abs/1603.07285.

[ES16] **Eldan, Ronen and Ohad Shamir**. "The power of depth for feedforward neural networks". In: *Workshop and Conference Proceedings*. Vol. 49. PMLR, 2016, pp. 1–34.

[Fey18] **Feydy, J.** *Automatic differentiation for applied mathematicians*. Tech. rep. ENS Paris et Paris Saclay, 2018. URL: https://www.ljll.math.upmc.fr/gtt/beamers/GTT_13-03-2018_Feydy.pdf.

[Fra+22] **Fragemann, Jana et al.** *Review of Disentanglement Approaches for Medical Applications: Towards Solving the Gordian Knot of Generative Models in Healthcare*. en. Mar. 2022. DOI: 10.36227/techrxiv.19364897.v1. URL: https://www.techrxiv.org/articles/preprint/Review_of_Disentanglement_Approaches_for_Medical_Applications_Towards_Solving_the_Gordian_Knot_of_Generative_Models_in_Healthcare/19364897/1.

[GB10] **Glorot, Xavier and Yoshua Bengio**. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[GBB11] **Glorot, Xavier, Antoine Bordes, and Yoshua Bengio**. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323. URL: https://proceedings.mlr.press/v15/glorot11a.html.

[GBC16] **Goodfellow, Ian, Yoshua Bengio, and Aaron Courville**. *Deep learning*. MIT press, 2016.

[GMH13] **Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton**. "Speech recognition with deep recurrent neural networks". en. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. Vancouver, BC, Canada: IEEE, May 2013, pp. 6645–6649. ISBN: 978-1-4799-0356-6. DOI: 10.1109/ICASSP.2013.6638947. URL: http://ieeexplore.ieee.org/document/6638947/.

[GSC00] **Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins**. "Learning to Forget: Continual Prediction with LSTM". en. In: *Neural Computation* 12.10 (Oct. 2000), pp. 2451–2471. ISSN: 0899-

7667, 1530-888X. DOI: 10.1162/089976600300015015. URL: https://direct.mit.edu/neco/article/12/10/2451-2471/6415.

[Hay04] **Haykin, Simon**. *Kalman filtering and neural networks*. John Wiley & Sons, 2004.

[Hay98] **Haykin, Simon**. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1998.

[He+15] **He, Kaiming et al.** "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". en. In: arXiv:1502.01852 (Feb. 2015). arXiv:1502.01852 [cs]. URL: http://arxiv.org/abs/1502.01852.

[HS15] **He, Kaiming and Jian Sun**. "Convolutional Neural Networks at Constrained Time Cost". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[HS97] **Hochreiter, Sepp and Jürgen Schmidhuber**. "Long Short-Term Memory". en. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1997.9.8.1735. URL: https://direct.mit.edu/neco/article/9/8/1735-1780/6109.

[KB15] **Kingma, Diederik P. and Jimmy Ba**. "Adam: A Method for Stochastic Optimization". en. In: *3rd International Conference for Learning Representations* arXiv:1412.6980 (Jan. 2015). arXiv:1412.6980 [cs]. URL: http://arxiv.org/abs/1412.6980.

[Kla+17] **Klambauer, Günter et al.** "Self-Normalizing Neural Networks". en. In: arXiv:1706.02515 (Sept. 2017). arXiv:1706.02515 [cs, stat]. URL: http://arxiv.org/abs/1706.02515.

[KM18] **Kim, Hyunjik and Andriy Mnih**. "Disentangling by factorising". In: *International Conference on Machine Learning*. PMLR, 2018, pp. 2649–2658.

[KSH17] **Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton**. "ImageNet classification with deep convolutional neural networks". en. In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386. URL: https://dl.acm.org/doi/10.1145/3065386.

[Kum17] **Kumar, Siddharth Krishna**. "On weight initialization in deep neural networks". en. In: arXiv:1704.08863 (May 2017). arXiv:1704.08863 [cs]. URL: http://arxiv.org/abs/1704.08863.

[LeC+98] **LeCun, Yann et al.** "Efficient BackProp". In: *Neural Networks: Tricks of the Trade*. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Vol. 1524. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. ISBN: 978-3-540-65311-0. DOI: 10.1007/3-540-49430-8_2. URL: http://link.springer.com/10.1007/3-540-49430-8_2.

[LH15] **Liang, Ming and Xiaolin Hu**. "Recurrent convolutional neural network for object recognition". In: *Proceedings of the IEEE con-*

ference on computer vision and pattern recognition. 2015, pp. 3367–3375.

[LJH15] **Le, Quoc V, Navdeep Jaitly, and Geoffrey E Hinton**. "A simple way to initialize recurrent networks of rectified linear units". In: *arXiv preprint arXiv:1504.00941* (2015).

[Mai99] **Maiorov, V.E**. "On Best Approximation by Ridge Functions". en. In: *Journal of Approximation Theory* 99.1 (July 1999), pp. 68–94. ISSN: 00219045. DOI: 10.1006/jath.1998.3304. URL: https://linkinghub.elsevier.com/retrieve/pii/S0021904598933044.

[MMD20] **Mouton, Coenraad, Johannes C. Myburgh, and Marelie H. Davel**. "Stride and Translation Invariance in CNNs". en. In: vol. 1342. arXiv:2103.10097 [cs]. 2020, pp. 267–281. DOI: 10.1007/978-3-030-66151-9_17. URL: http://arxiv.org/abs/2103.10097.

[Mni+16] **Mnih, Volodymyr et al.** "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[MP43] **McCulloch, Warren S. and Walter Pitts**. "A logical calculus of the ideas immanent in nervous activity". en. In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 0007-4985, 1522-9602. DOI: 10.1007/BF02478259. URL: http://link.springer.com/10.1007/BF02478259.

[Pér+03] **Pérez-Ortiz, Juan Antonio et al.** "Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets". In: *Neural Networks* 16.2 (2003), pp. 241–250.

[Pey20] **Peyré, Gabriel**. *Course notes on Optimization for Machine Learning*. Notes de cours de l'École Normale Supérieure. 2020. URL: https://mathematical-tours.github.io.

[PMB13] **Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio**. "On the difficulty of training recurrent neural networks". In: *International conference on machine learning*. Pmlr, 2013, pp. 1310–1318.

[Pol64] **Polyak, B.T.** "Some methods of speeding up the convergence of iteration methods". en. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (Jan. 1964), pp. 1–17. ISSN: 00415553. DOI: 10.1016/0041-5553(64)90137-5. URL: https://linkinghub.elsevier.com/retrieve/pii/0041555364901375.

[ROS57] **ROSENBLATT, MURRAY**. "Some Purely Deterministic Processes". In: *Journal of Mathematics and Mechanics* 6.6 (1957), pp. 801–810. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900623.

[Rou72] **Rousseau, Jean-Jacques**. *Discours sur l'origine et les fondemens de l'inégalité parmi les hommes*. Vol. 1. chez Marc Michel Rey, 1772.

[Sal+18] **Salehinejad, Hojjat et al.** "Recent Advances in Recurrent Neural Networks". en. In: arXiv:1801.01078 (Feb. 2018). arXiv:1801.01078 [cs]. URL: http://arxiv.org/abs/1801.01078.

[Smi17] **Smith, Leslie N.** "Cyclical Learning Rates for Training Neural Networks". en. In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. Santa Rosa, CA, USA: IEEE, Mar. 2017, pp. 464–472. ISBN: 978-1-5090-4822-9. DOI: 10.1109/WACV.2017.58. URL: http://ieeexplore.ieee.org/document/7926641/.

[SP97] **Schuster, Mike and Kuldip K Paliwal**. "Bidirectional recurrent neural networks". In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[Sut+13] **Sutskever, Ilya et al.** "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[TH12] **Tieleman, Tijmen and Geoffrey Hinton**. "Lecture 6.5-rmsprop, coursera: Neural networks for machine learning". In: *University of Toronto, Technical Report* 6 (2012).

[Via22] **Vialle, Stéphane**. *Séquence Thématique ST7-76 - Similation à haute performance (2021-2022)*. "Ingénieur" Curriculum, CentraleSupélec. 2022.

[Wer90] **Werbos, P.J.** "Backpropagation through time: what it does and how to do it". en. In: *Proceedings of the IEEE* 78.10 (Oct. 1990), pp. 1550–1560. ISSN: 00189219. DOI: 10.1109/5.58337. URL: http://ieeexplore.ieee.org/document/58337/.

[Whi51] **Whittle, Peter**. *Hypothesis testing in time series analysis*. Vol. 4. Almqvist and Wiksells boktr., 1951.

[Wil92] **Williams, Ronald J**. "Training recurrent networks using the extended Kalman filter". In: *International joint conference on neural networks*. Vol. 4. Citeseer, 1992, pp. 241–246.

[WJ16] **Wang, Shuohang and Jing Jiang**. "Learning Natural Language Inference with LSTM". en. In: arXiv:1512.08849 (Nov. 2016). arXiv:1512.08849 [cs]. URL: http://arxiv.org/abs/1512.08849.

[Xue+10] **Xuemei, Li et al.** "Hybrid support vector machine and ARIMA model in building cooling prediction". In: *2010 International Symposium on Computer, Communication, Control and Automation (3CA)*. Vol. 1. 2010, pp. 533–536. DOI: 10.1109/3CA.2010.5533864.

[YK15] **Yu, Fisher and Vladlen Koltun**. "Multi-Scale Context Aggregation by Dilated Convolutions". en. In: arXiv:1511.07122 (Apr. 2015). arXiv:1511.07122 [cs]. URL: http://arxiv.org/abs/1511.07122.

[Zei+10] **Zeiler, Matthew D. et al.** "Deconvolutional networks". In: *2010 IEEE Computer Society Conference on Computer Vision and Pat-*

tern Recognition. 2010, pp. 2528–2535. DOI: 10.1109/CVPR.2010.5539957.

[Zei12] **Zeiler, Matthew D.** "ADADELTA: An Adaptive Learning Rate Method". en. In: arXiv:1212.5701 (Dec. 2012). arXiv:1212.5701 [cs]. URL: http://arxiv.org/abs/1212.5701.