Università degli Studi di Milano-Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di Laurea Magistrale in Data Science

TRANSFER LEARNING AND HYPERPARAMETER OPTIMIZATION FOR AUTOMATIC CAR DAMAGE DETECTION

Relatore: Antonio Candelieri

Tesi di Laurea Magistrale di:

Raffaele Anselmo

Matricola 846842

Anno Accademico 2019-2020

Alla nonna Giovanna.

“Happiness is an enemy. It weakens you.
Suddenly, you have something to lose."
Niki Lauda

Acknowledgements

Sebbene a tagliare il traguardo sia una persona sola, la vittoria è sempre della squadra. In questo percorso ho potuto contare su delle persone meravigliose che non mi hanno fatto mai mancare il loro supporto e a loro tutti vanno i miei ringraziamenti. Sono stati due anni pieni di avventure, sfide ed emozioni che porterò per sempre nel cuore.

Vorrei ringraziare in primis il mio relatore, Prof. Antonio Candelieri, per aver contribuito a farmi appassionare ai temi di machine learning e per le preziose indicazioni che mi ha fornito in questi mesi.

Ringrazio infinitamente mia mamma, la persona più forte che conosca, mio cugino Corrado e tutta la mia famiglia per avermi supportato ogni giorno.

Un ringraziamento particolare ai miei zii e cugini milanesi, che mi hanno sempre fatto sentire a casa.

Ad Alessandra, rifugio dei miei pensieri e complice della mia felicità.

Ai Carli, con i quali neanche in questi due anni abbiamo festeggiato la champions. Ci rifaremo, a costo di tornare a guardare le partite a terra in salone.

A Lorenzo, perchè essere compagni di viaggio quando c'è il sole è facile, con il brutto tempo però vale di più. A Vincenzo, che ha rivoluzionato la mia quarantena con il biglietto per Verdansk e a tutte le triglie incontrate con il team. A tutti i miei amici, perchè sono i peggiori al mondo e per questo gli voglio bene da morire.

Infine i ringraziamenti più importanti vanno alla nonna Giovanna e alla zia Gigia, esempi di amore incondizionato. Vi porterò per sempre con me.

Summary

This thesis focuses on the hyperparameter optimization problem applied on convolutional neural networks with transfer learning for image classification. Since the convolutional neural networks have been acknowledged as the *state-of-the-art* solution for image classification, several studies focused on *optimizing* their performances. Architecture, training, image preprocessing are only some examples of the deep learning workflow components that have been developed in order to improve the convolutional neural networks capabilities. The first chapter deals with image classification, providing an historical background and an overview of convolutional neural networks and transfer learning.

Although these paradigms are well-defined on benchmark datasets, the application of convolutional neural networks to real world examples requires still too many decisions without any a priori information about their goodness. In the deep learning context all the decisions required in the model building are referred as *hyperparameters* of the model and the problem of choosing the best set of hyperparamters is known as *hyperparameter optimization*. The second chapter deepens this problem, discussing some techniques along with their strengths and weaknesses.

Lastly, the application of a promising hyperparamter optimization technique, BOHB, is applied to a convolutional neural network from scratch and on a transfer learning model. The resulting best model configuration is used to develop a business solution for insurance market: an automated suite for the damage detection on cars.

Contents

Chapter 1

Image Classification

Image classification is one of the first tasks that have been addressed in computer vision. In more than 50 years of research several techniques have been developed, such as hierarchical models and local feature-based methods. The turning point was in 2012, when the convolutional neural networks definitively took the stage. These networks showed astonishing performances when compared to previous methods; furthermore they have been used also for other tasks, e.g. image segmentation. The increasing demand of new solutions based on these intelligent systems addressed the need to reuse the already learned algorithms. This knowledge transfer has been formalized in transfer learning paradigms.

In this chapter an historical background on image classification and computer vision is provided, along with an overview of convolutional neural networks and transfer learning approaches on image classification.

1.1 Background

Artificial visions systems have always fascinated humans since pre-historic times. The earliest mention about an artificial visually-guided agent, recovered by Andreopoulos et Tsotsos (2013) [1], appeared in classical mythology: a bronze giant

named Talos was created by the ancient god Hephaestus and was given as a gift to King Minos of the Mediterranean island of Crete [2]. According to legend, the robot served as a defender of the island from invaders by circling it three times a day and making sure that the laws of the land were upheld by the island's inhabitants.

The hype behind these artificial vision systems didn't hold back during the decades; on the contrary, its domain shifted from myths to scientific papers thanks to the developments in science and technology.

1.1.1 The first steps

In the late 1950s one of the most influential papers in Computer Vision has been published by two neurophysiologists, David Hubel and Torsten Wiesel. Their publication [3], entitled *"Receptive field of single neurons in the cat's striate cortex"*, aimed to investigate how the neural activity of the cat can respond to some visual stimolus. Their results were initially fruitless, but after few months of research they noticed that one neuron fired as they were showing a new image to the cat [4] (see Fig. 1.1).

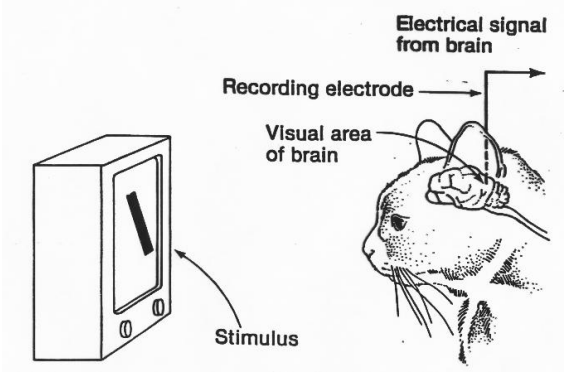

Figure 1.1: Hubel and Wiesel experiment. Image source [5]

This insight carried out the pillars of image classification (and, more generally, deep learning): the brain is made of simple and complex neurons and the visual processing always starts with simple structures such as oriented edges.

New disciplines come from new technologies. Computer vision was enabled by Russel Kirsch and his colleagues' studies, who developed the first digital image scanner in 1957. One of the first photographs scanned (Fig. 1.2), a picture of Kirsch's three-months-old son, was captured as just 30,976 pixels, a 176 × 176 array, in an area 5 cm × 5 cm. The bit depth was only one bit per pixel, the colors stark black and white with no intermediate shades of gray, but, by combining several scans made using different scanning thresholds, grayscale information could also be acquired. They used the computer to extract line drawings, count objects, recognize alphanumeric characters, and produce oscilloscope displays [6].

Figure 1.2: Russell Kirsch's son in one of the first digitally scanned image ever (1957). Portland, Art Museum. Image source [6]

At those times the computer vision paradigm was stated by Lawrance Roberts (1963). In his Ph.D. thesis, *"Machine perception of three-dimensional solids"*, he described the process of deriving 3D info about solid objects from 2D photographs. The goal of the program was to process 2D photographs into line drawings, then build up 3D representations from those lines and, finally, display 3D structures of objects with all the hidden lines removed [7]. Later, Roberts joined DARPA and is now known as one of the inventors of internet.

Early systematic attempts on vision systems were also traced to the Hitachi labs in Japan, where the term *machine vision* originated so as to distinguish its

pragmatic goal of constructing practical applications, compared to the more general term *computer vision*, popularly used to also include less pragmatic goals [1].

1.1.2 The birth of CNNs and the local feature-based methods

During the 1900s, computer vision systems architecture followed a *passive* approach. As argued by Marr (1982) a bottom-up approach was used, with no control over the data acquisition process. Based on the ideas of Hubel and Wiesel, he also introduced a framework for vision where low-level algorithms that detect edges, curves and corners were used as stepping stones towards a high-level understanding of visual data.

Hubel and Wiesel continued to inspire further studies, especially the ones conducted by Kuniko Fukushima (1980). He built a hierarchical, multilayered artificial neural network of simple and complex cells that could recognize patterns without being affected by any position shift. The network was named *Necognitron* [8] and served as the inspiration for convolutional neural networks (CNNs) because of the emergence of convolutional layers (Fig. 1.3). The simple cells extract the local features, while their deformations, suchs as local shifts, are tolerated by the complex cells. The intermediate layers of the network were trained using *AiS* (*Add-if-Silent*) learning rule. Under the AiS rule, a new cell is generated and added to the network if all postsynaptic cells are silent in spite of non-silent presynaptic cells. The generated cell learns the activity of the presynaptic cells in one-shot. Once a cell is generated, its input connections do not change any more [9].

While in the late 1980s researches in computer vision shifted to more mathematically oriented approaches that were significantly different from the classical AI techniques of the time, in recent works the role role of learning algorithms has become much more important. This resulted in a blurring of the distinction between

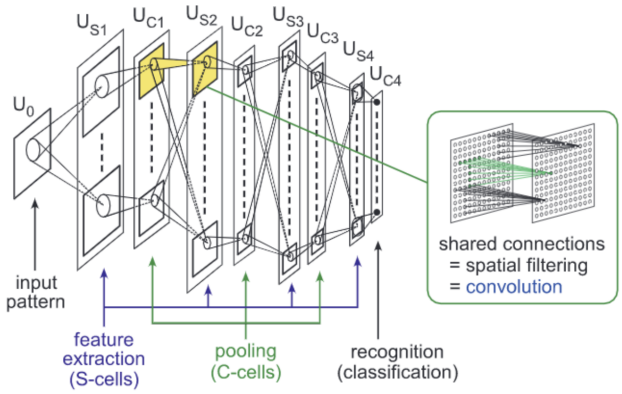

Figure 1.3: Necognitron architecture. Image source [9]

computer vision and AI. Often the main innovation presented in the computer vision papers were more closely related to machine learning, while vision were only treated as a small after-effect/application of the presented learning-based algorithm [1].

A decade after Fukushima's Necognitron, Yann LeCun (1989) substituted the AiS rule of Necognitron with a backpropagation algorithm, which was applied for the first time to convolutional neural networks. Its network was named *LeNet* [10]. He conducted the experiments on handwritten zip code recognition providing the first version of the famous MNIST dataset [11].

Although the backpropagation algorithm was efficient on updating the network's weights, the convolutional neural networks were not mature enough for several reasons, among which the limited computational capacity of the time and the lack of sizeable datasets.

As the developments for CNNs continued behind the scenes, the second half of 1990s was dominated by local feature-based recognition methods. An early local feature-based approach to recognition was proposed by Rao and Ballard in 1995 [12]. They aimed to extract local feature vectors encoding the multiscale local orientation of various image locations.

Four years later Lowe published what is considered the most important paper for

feature-based methods and it is still one of the most cited computer vision paper ever. Lowe presented the SIFT algorithm [13], a visual recognition system that uses local features that are invariant to rotation, location, and, partially, changes in illumination. Several further works were influenced by Lowe's approach, until the CNNs took the stage.

1.1.3 AlexNet: the breakthrough

After 50 years of studies, computer vision gained the attention of many researchers from all over the world. One of the main problems of the 1980s was the lack of an institution able to collect all the studies and indicate the new areas of research. To overcome this problem, in 2005 the Pascal VOC (Visual Object Classes) project [14] was launched. It ran annual challenges from 2005 to 2012 and provided standardised image datasets for object class recognition. Such a stable and controlled environment allowed to mark some baselines and to compare the different methods proposed. The first competition counted 1578 images related to only 4 classes: bicycles, cars, motorbikes and people, but its complexity increased during the years, reaching more than 10,000 images with 20 annotated classes.

Following the Pascal footsteps, in 2010 the Imagenet [15] project started the Imagenet Large Scale Visual Recognition Challenge (ILSVRC). As for Pascal, it run annual challenges, followed by workshops, but giving a larger database, with more than 14 millions images across over 10,000 classes.

The first two editions of ILSVRC didn't give any noticeable improvement on image classification methods, but in 2012 the revolution was ready to come.

On September 30th 2012, a team from the University of Toronto introduced a convolutional neural network model, called *AlexNet* [16], that achieved a top-5 error of 15.3%, more than 10.8 percentage points lower than that of the runner up and of the former editions' winners [17].

Although AlexNet took the same top-down approach of former CNNs, where

successive filters are designed to capture more and more subtle features, it leveraged their full potential for 4 reasons:

1. **ReLU Nonlinearity**: it introduced better non-linearity in the network with the *ReLU* activation functions $f(x) = \max(0, x)$, whose derivative is 0 if the feature is less than 0 and 1 for positive values. The use of ReLU gave a strong speed-up to the network training with respect to the *tanh* and *sigmoid* functions, the most common at the time.

2. **Training on multiple GPUs**: the network was spread across two GPUs, exploiting the GPUs' ability of reading and writing to each other memory directly, without going through host machine memory. The parallelization scheme used by Krizhevsky *et al.* put half of the neurons on each GPU and let the GPUs communicate only in certain layers. Although the use of GPU does not provide a gain in terms of performance itself, it enabled the training of deeper nets.

3. **Dropout**: at every epoch a percentage of neurons is masked (*dropped out*), so it does not contribute to the forward level and it is excluded from back-propagation. This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. Therefore, it is forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons [16].

4. **Data Augmentation**: when fed to the network, images are shown with random translation, rotation, crop. In so doing, the network is more aware of the images attributes, rather than the images themselves [18].

AlexNet was the first network designed to be *deep*. Thanks to the use of multiple GPUs more convolutional layers have been stacked before pooling operations, allowing to capture finer features that revealed to be useful for classification. As

proved by the authors [16], removing any of the middle layers results in a loss of about 2% for the top-1 performance of the network. This showed how much the depth is important to achieve the best performance.

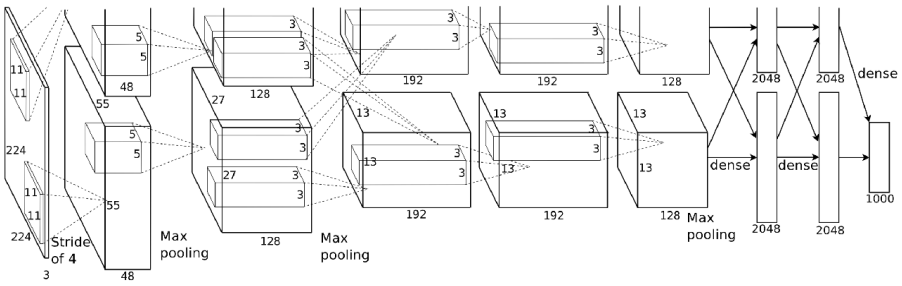

Figure 1.4: AlexNet architecture. Image source [16]

1.1.4 Deeper is better: VGGNet

The ImageNet Challenge 2014 showed an other important step forward when the Visual Geometry Group of Oxford University submitted *VGG16*. While previous derivatives of AlexNet focused on smaller window sizes and strides in the first convolutional layer, VGG addresses the main characteristic of AlexNet: depth [19].

While *"VGG"* takes its name from the team, *"16"* stands for the number of training layers of the network. In the same paper an extended version, VGG19, was proposed, but with no relevant increase in terms of performances.

From a computational point of view, the main characteristic of VGG is the presence of only 3x3 convolutions, that is explained in Section 1.4.6.

VGG achieved second place in the 2014 ImageNet competition with its top-5 error of 7.3%, which they decreased to 6.8% after the submission. The architecture of VGG16 is detailed in Section 1.3.6.

1.1.5 The last trends: the connection matters

VGG is still a *state-of-the-art* (SOTA) network for image classification, but also different interesting approaches have been proposed in the last years.

GoogleNet [20] was proposed the same year of VGG16 and its architecture is based on finding out how an optimal local sparse structure in a convolutional vision network can be approximated and covered by readily available dense components. This is obtained by stacking one module of layers over another inside the network, giving multiple auxiliary classifiers. Obiouvsly, more classifiers inside the network means more computational costs. To overcome this effort, the Google team used 1x1 convolutions, able to reduce the dimensionality of the features and at the same time combine the feature maps in a way that can be beneficial from a representation perspective. The best GoogleNet ensemble achieved a 6.7% error on ImageNet.

The general trend of going deeper quickly faced a serious problem: the *vanishing gradient*. Actually, at some point, stacking more layers does not lead to better performance, instead they decrease.

When the gradient has to backpropagate from the last layer to the first, it can completely vanish somewhere along the way (hence the name *vanishing*), especially when many layers are involved. Residual or shortcut connections were introduced by the *ResNet* models [21], which created alternate paths for the gradient to skip the middle layers and directly reach the initial layers. It is still one of the best performing network on ImageNet, with a 3.6% top-5 error rate.

In opposite to the general trend of going deeper, Google also aimed to create lighter CNN models, able to run in devices with extreme memory and computation constraints (known as edge-devices). *MobileNet* [22] introduced the idea of separable convolutions: it breaks down a 2D convolution kernel into two separate convolutions, depthwise and pointwise [23].

1.2 Convolutional Neural Networks

Convolutional networks stand out as an example of neuroscientific principles influencing deep learning as they take inspiration from Hubel and Wiesel experiments.

Since AlexNet entered the ImageNet Competition in 2012, all the successive editions have been dominated by convolutional neural netwoks, making them ubiquitous on image classification.

CNNs were some of the first neural networks to solve business applications and they also initiated the commercial interest around deep learning.

Convolutional networks are not so different from classical feed-forward neural networks, instead they provide a way to specialize neural networks to work with data that has a clear grid-structured topology and to scale such models to very large size.

Two-dimensional data, like black & white images, is one use case, but CNNs have been applied also to one-dimensional data. Lang and Hilton (1988) [24] used CNNs to train one-dimensional time series. They also introduced the use of back-propagation as training algorithm, which approach was followed a year later by LeCun *et al.* (1989) to train 2-D convolution applied to images.

However, to process one-dimensional, sequence data, the Recurrent Neural Networks (RNNs) are now preferred [25].

1.2.1 Architecture

A simple convolutional network (or ConvNet) is made of a sequence of layers, and every layer transforms one volume of activations to another through a differentiable function.

Three types of layers are used to form a full ConvNet architecture: Convolutional Layer, Pooling Layer and Fully-Connected Layer. Stacking sequentially these layers, ConvNets transoform the original image layer by layer from the pixel values

to the final class scores.

The individual layers are described in the following section.

Convolutional Layer

The convolutional layer is the building box of a convolutional neural network. It applies the convolution operation and it is responsable of most of the computational heavy lifting.

Given a function $k(\cdot)$ called *kernel* and applying this function to an input i spaced in an axis t, the convolution operation is defined as the integral of the product of the two functions after one is reversed and shifted [26]. It is typically denoted by an asterisk:

$$s(t) = (i * k)(t). \tag{1.1}$$

The output $s(t)$ is sometimes referred as *feature map*.

In machine learning applications, the input is usually a multidimensional array of discrete data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. Convolutions over more than one axis at a time are defined as follows:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{1.2}$$

Convolution is commutative, meaning we can equivalently write:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{1.3}$$

The commutative property of convolution is obtained flipping the kernel relative to the input, in the sense that as m increases, the index into the input increases but the index into the kernel decreases. While the commutative property is useful

for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{1.4}$$

Discrete convolution can be viewed as multiplication by a matrix (Figure 1.5) which elements must respect some constraints.

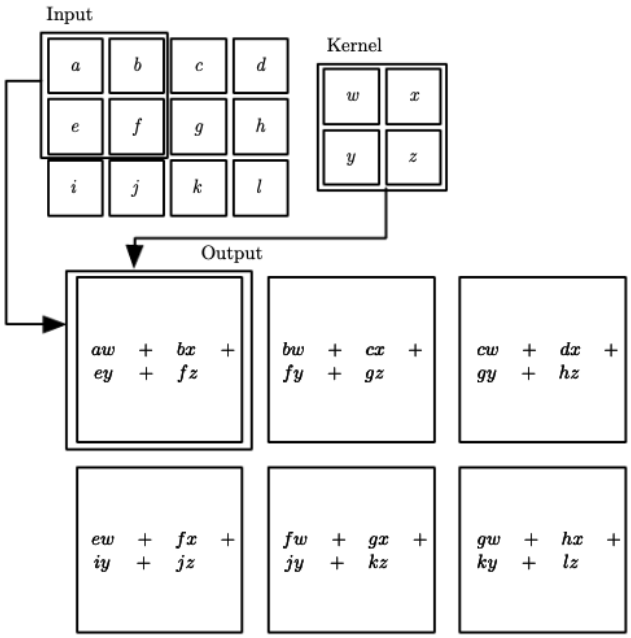

Figure 1.5: An example of 2-D convolution. Image source [25]

The convolutional layer consists of a set of learnable kernels that takes the name of learnable filters. Each filter is usually small spatially (along width and depth), but extends through the full depth of the input volume. When the filter is slided over the the width and the depth of the input volume it will produce a two-dimensional activation map that gives the responses of that filter at every spatial position. Every entry in the output volume is thus an output of a neuron

that "looks" only at a small region of the input. This results in a **local connectivity** that is one of the main motivation of using CNNs in place of classical artificial neural networks. Actually, if there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime.

Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network [27].

The use of small filters is justified by the assumption that, even though the image might have thousands or million of pixels, we want to detect small, meaningful features that occupy few pixels.

The spatial extent of this connectivity is an hyperparameter called *receptive field* F of the neuron (equivalently this is the filter size).

The output volume size is determined by the receptive field, along to other three hyperparameters: *depth*, *stride* and *zero padding*.

- The *depth* of the output volume corresponds to the number of filters used.
- The *stride* determines how the filter is slided. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 then the filters jump 2 pixels at a time as we slide them around. Greater stride will produce smaller output volumes spatially.
- The *zero-padding* is often used to preserve the spatial size of the input volume so the input and output width and height are the same.

Given an input volume of dimension $W_1 \cdot H_1 \cdot D_1$, the number of filters (K), the receptive field (F), the stride (S) and the amount of zero padding used on the

border (P), the output volume will have dimension $W_2 \cdot H_2 \cdot D_2$, where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$
- $D_2 = K$

However, these spatial hyperparameters have to respect some integer contraints for the output volume. Actually, a wrong set of receptive field and stride would result in a float output dimension, which is invalid.

Another main characteristic which made CNNs so useful is the **parameter sharing** schema used in convolutional layers to control the number of parameters. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. In a convolutional network, each member of the kernel is used at every position of the input. This means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation but it does further reduce the storage requirements of the model to k parameters [25].

To make this more clear, let's use AlexNet as example. The Krizhevsky *et al.* architecture accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field $F = 11$, stride $S = 4$ and no zero padding $P = 0$. Since $(227 - 11)/4 + 1 = 55$, and since the Conv layer had a depth of $K = 96$, the Conv layer output volume had size [55x55x96]. This conv layer would have $55 \cdot 55 \cdot 96 = 290400$ neurons, and each has $11 \cdot 11 \cdot 3 = 363$ weights and 1 bias. Together, this adds up to $290400 \cdot 364 = 105{,}705{,}600$ parameters on the first layer of the ConvNet alone [27]. Clearly, this number is too high.

Instead, using the parameter sharing scheme, the neurons in each depth slice will have the same weights and bias. So, the first convolutional layer of AlexNet have only 96 unique set of weights (one for each depth slice) for a total of $96 \cdot 11 \cdot *11 \cdot 3$

unique weights that, adding the 96 biases, will result in 34,944 parameters, that is a more reasonable number.

Lastly, the particular form of parameter sharing causes the layer to have a property called equivariance to translation. When processing images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations [25].

Convolutions themselves are not naturally equivariant to some other translations, such as changes in the rotation or scale of an image. However, a CNN as a whole can learn filters that fire when a pattern is presented at a particular orientation [28]. In order to obtain a CNN fully equivariant to transformation, some special mechanisms are required, such as augmenting the training data with transformed images for scale, rotation, brightness, zoom, and so on.

Pooling Layer

Once performed the convolutions on the input images, we obtain the activation maps. In a successive stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation (ReLU) function. This stage is sometimes called the detector stage. Finally, a pooling function is applied to modify and reduce the output of the layer (Fig. 1.6).

Pooling layers apply fixed functions, so they do not contain any parameter. The most common functions are max or average pooling. The first uses the maximum value of a neighborhood to reduce the volume dimension, while the latter uses the mean. Max pooling is typically done in the middle of the CNN architecture to reduce spatial size, whereas average pooling is normally used as the final layer of the network (as in GoogLeNet, SqueezeNet, ResNet) where we wish to avoid using fully-connected layers entirely [28].

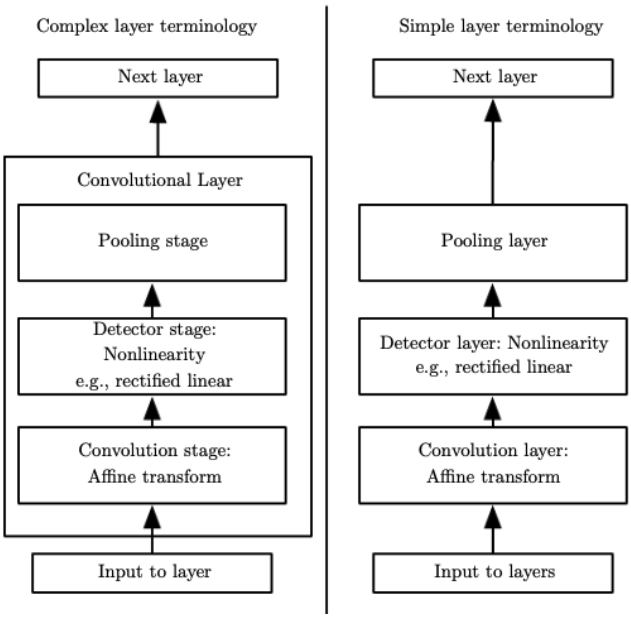

Figure 1.6: The components of a typical convolutional neural network layer in the two commonly used terminology for describing the layers. (Left) The convolutional layer is viewed as the combination of the convolution operation, non-linearity activation and poolin stage. (Right) Each stage, convolution, detection and pooling is viewed as single layers. Image source [25]

When applying pooling operations, two hyperparameters, the *receptive field* (F) and the *stride* (S) , control the output dimension dimension $W_2 \cdot H_2 \cdot D_2$ in the same way as done by the convolution operation, but without the zero-padding and keeping the same depth as the previous layer.

Precisely because of this similarity with convolution's downsampling, Springenberg *et al.* [29] suggested to avoid the pooling layers for being found redundant and useless. Also, pooling can complicate some kinds of neural network architectures that use top-down information, such as Boltzmann machines and autoencoders [25].

Fully-connected layers

The last block of a ConvNet is made of fully-connected layers (or *dense* layers). Fully-connected layers are composed of neurons that perform a weighted sum of

all the inputs, weighted by the *weights* of the connections from the inputs to the neuron, to which is added a *bias*. This weighted sum is then passed through an activation function to produce the output [30].

In opposite to convolutional layers, in f-c layers every neuron in one layer is connected to every neuron in the next layer (hence the name *fully-connected*).

These layers make the predictions based on the features extracted by the convolutional block.

In Figure (1.7) the three main components, Convolutional layers, pooling layers and fully-connected layers are combined in an example ConvNet architecture.

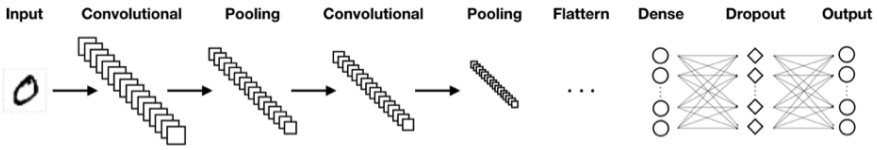

Figure 1.7: Example of ConvNet architecture

1.2.2 Learned Features

The intuition of Hubel and Wiesel, whose argued that the brain is made of simple and complex cells, found a visual representation through the features learned from modern deep convolutional neural networks.

Independently on the task and the training data, deep CNNs share a curious virtue: they all tend to learn general features, like Gabor filters or color blobs [31], in the first layers. The, they reassamble these lower-level features into higher-lever, more specific, features the last layers of the network.

This hierarchical feature representation can be seen in figure (1.8). When processing face's images, the first layers identifies edges and blobs; the second layer discriminates more higher-level features, like noses, eyes and mouths; the last layer combines these feature to recognize faces.

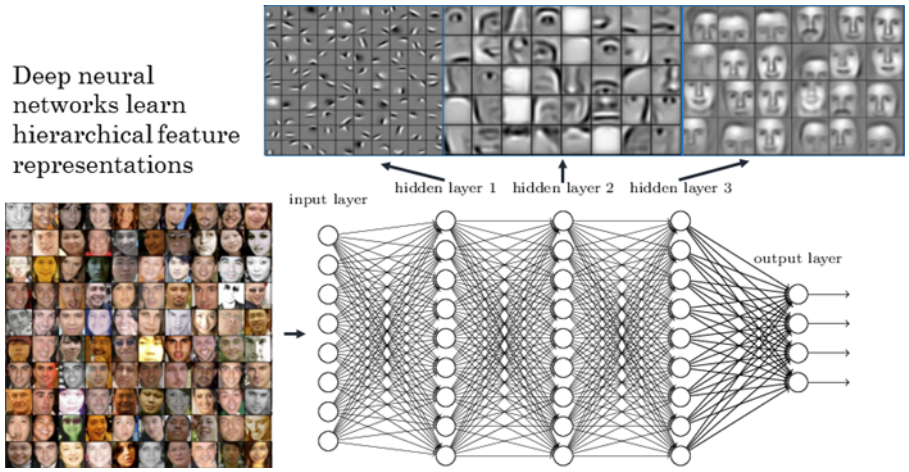

Figure 1.8: CCN hierarchical feature representations. Image source [32]

1.2.3 Region Based Convolutional Neural Neworks

Convolutional neural networks have been applied to images also for other tasks than image classification.

Region Based Convolutional Neural Networks (R-CNNs) are derived from classical CNNs in order to detect, classify and, most important, localize the objects in images. They are used for the task of semantic or instance segmentation. Semantic segmentation detects all the objects present in an image at the pixel level and outputs regions with different classes or objects. In semantic segmentation the pixels are grouped in a semantically meaningful way, so pixels belonging to different objects (such as persons, cars or buildings) are grouped separately.

Differently, semantic instance segmentation (or instance segmentation for brevity) goes deeper, identifying not only the object classes, but also each instance for each object.

Both these methods are subsequent to the object identification process, i.e. the detection of objects in an image. The output difference between semantic and instance segmentation is shown in Figure 1.9.

(a) Input Image (b) Semantic segm. (c) instance segm.

Figure 1.9: Semantic segmentation (b) makes no difference between instance in “ballon" class, while instance segmentation (c) identify each object of the same class separately. Image source [33]

The first *R-CNN* was proposed by Girshick *et al.* in 2014 [34] with the aim to extend the CNN results achieved on Imagenet Challenge to the object detection PASCAL VOC Challenge. This method is composed by an algorithm, selective search, that extract 2000 proposed regions to be evaluated. Each region is fed into a CNN that produces a 4096-dimensional feature vector, used by SVMs to classify the presence of the object within the candidate region proposal in the last stage (Figure 1.10).

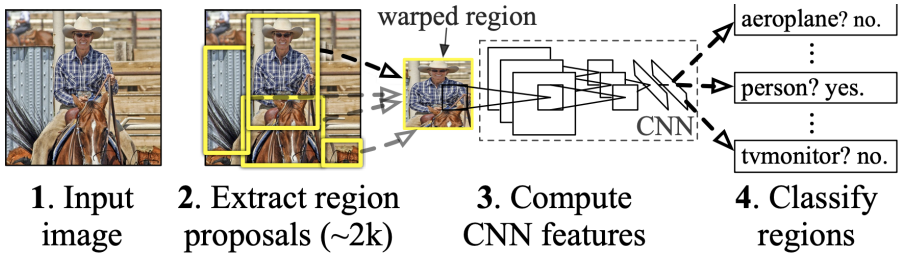

Figure 1.10: R-CNN pipeline. Image source [34]

The problems behind this approach reside in the computational complexity and inefficiency of the selection algorithm. In fact, all the 2000 region proposal per image must be evaluated and classified, a process that requires almost 50 seconds for image at inference stage [35]. Furthermore, the selection algorithm is fixed and so it is not able to learn which regions to propose. This could lead to the generation of bad candidate region proposals.

Girshick surpassed the main computational difficulties by reverting the process of region proposal and feature extraction. Rather than first operate the region proposal and then extract the features with a CNN, the proposed approach, *Fast R-CNN* [36], first extract the features from the entire image and then it uses the feature map to identify the region of proposals. These proposals are warped into squares and reshaped into a fixed size through a RoI pooling layer to be finally computed by a fully-connected layer. This pipeline results to be 9x faster at training-time and 213x faster at test-time compared to R-CNN.

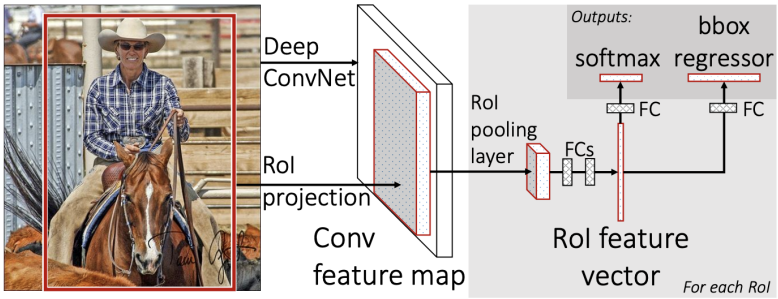

Figure 1.11: Fast R-CNN pipeline. Image source [36]

However, neither R-CNN nor Fast R-CNN modified the selection algorithm, which remained slow and dummy. Ren *et al.* [37] substituted selective search with a trainable fully convolutional network, Region Proposal Network (RPN), that simultaneously predicts object bounds and objectness scores at each position. This new framework, *Faster R-CNN*, increased the performances reducing the training and testing time even more.

Finally, a group of researchers at Facebook, extended the capabilities of Faster R-CNN by generating a high-quality segmentation mask for each instance detected. This is *Mask R-CNN*.

The architecture of Mask R-CNN [38] (Fifure 1.12 is almost the same of Faster R-CNN, composed of a feature extractor CNN (called *backbone*) connected to a network responsible of the bounding-box recognition (called *head*). In addition, Mask R-CNN adds a branch to the *head* for predicting segmentation masks on

each Region of Interest (RoI). The parallelization of the two branches of the head only adds a small computational overhead.

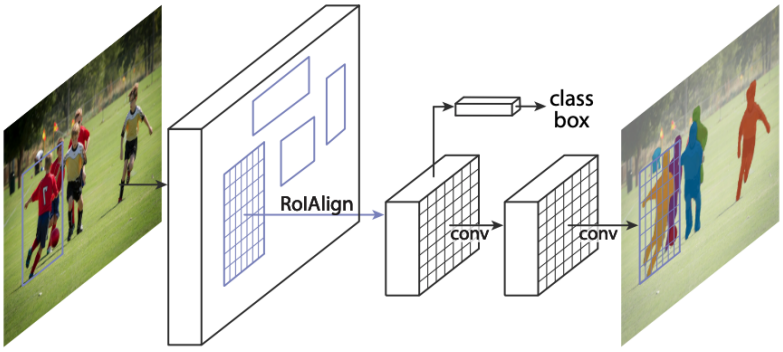

Figure 1.12: Mask R-CNN framework for instance segmentation. Image source [38]

A mask encodes an input object's spatial layout, so a perfect pixel-to-pixel alignment between the network inputs and outputs is required. Fast R-CNN is not able to do it, because of the RoiPool, that performs coarse spatial quantization for feature extraction. Although this misalignment may not impact classification, which is robust to small translations, it has a large negative effect on predicting pixel-accurate masks.

To fix this misalignment, a RoiAlign layer is proposed. It avoids any quantization of the RoI boundaries or bins by using bilinear interpolation to compute the exact values of the input features at four regularly sampled locations in each RoI bin.

1.3 Transfer Learning

Humans have an inherent ability to transfer knowledge across the task. Learning to play chess will help to play dama, but it also improves the ability of problem-solving. This example shows how the knowledge transfer could apply both if the two domains are the same or even if they are different. What we acquire as knowledge while learning about one task, we utilize in the same way to solve related tasks.

Since Neural Information Processing Systems (NIPS) 1995 workshop *Learning to*

Learn: Knowledge Consolidation and Transfer in Inductive Systems [39], machine learning community posed its attention to apply the above human behaviour to machines. Several terms, such us *Learning to Learn, Knowledge Consolidation* and *Inductive Transfer* have been used interchangeably referring to this knowledge transfer, but it is now known as *Transfer Learning*, as suggested by Goodfellow [25], who refer to transfer learning as the "Situation where what has been learned in one setting is exploited to improve generalization in another setting".

A slightly different technique related to transfer learning is the multi-task learning framework, which tries to learn multiple tasks simultaneously even when they are different.

Rather than learning all the source and target tasks simultaneously (often uncovering common features that can benefit each individual task), transfer learning focuses on transfer knowledge across tasks. So, the roles of source and target tasks are no longer symmetrical.

1.3.1 Notations and Definitions

In this section, the notations and definitions used are taken by Qian Yang *et al.* [40].

A domain $\mathcal{D}$ consists of two components: a feature space $\mathcal{X}$ and a marginal probability distribution $P(X)$, where $X = \{x_1, \ldots, x_n\} \in \mathcal{X}$.

Given a specific domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$, a task consists of two components: a label space $\mathcal{Y}$ and an objective predictive function $f(\cdot)$ (denoted by $\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$, which is not observed but can be learned from the training data, which consist of pairs $\{x_i, y_i\}$, where $x_i \in X$ and $y_i \in Y$. The function $f(\cdot)$ can be used to predict the corresponding label, $f(x)$, of a new instance x. From a probabilistic viewpoint, $f(x)$ can be written as $P(y|x)$.

Let's now discuss the case when there is one source domain $\mathcal{D}_S$ and one target domain $\mathcal{D}_T$. The source domain data are denoted as $D_S = \{(x_{S_1}, y_{S_1}), \ldots, (x_{S_{n_S}}, y_{S_{n_S}})\}$,

where $x_{S_i} \in \mathcal{X}_S$ is the data instance and $y_{S_i} \in \mathcal{Y}_S$ is the corresponding class label. Similarly, the target domain data are denoted as $D_T = \{(x_{T_1}, y_{T_1}), \ldots, (x_{T_{n_T}}, y_{T_{n_T}})\}$, where $x_{T_i} \in \mathcal{X}_T$ is the data instance and $y_{T_i} \in \mathcal{Y}_T$ is the corresponding class label. Usually the target domain has much less data than the source domain, so $0 \leq n_T << n_S$.

Based on these notations, the Transfer learning is defined as follows:

Given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, transfer learning aims to help improve the learning of the target predictive function $f_T(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$.

A transfer learning process is illustrated in Figure (1.13).

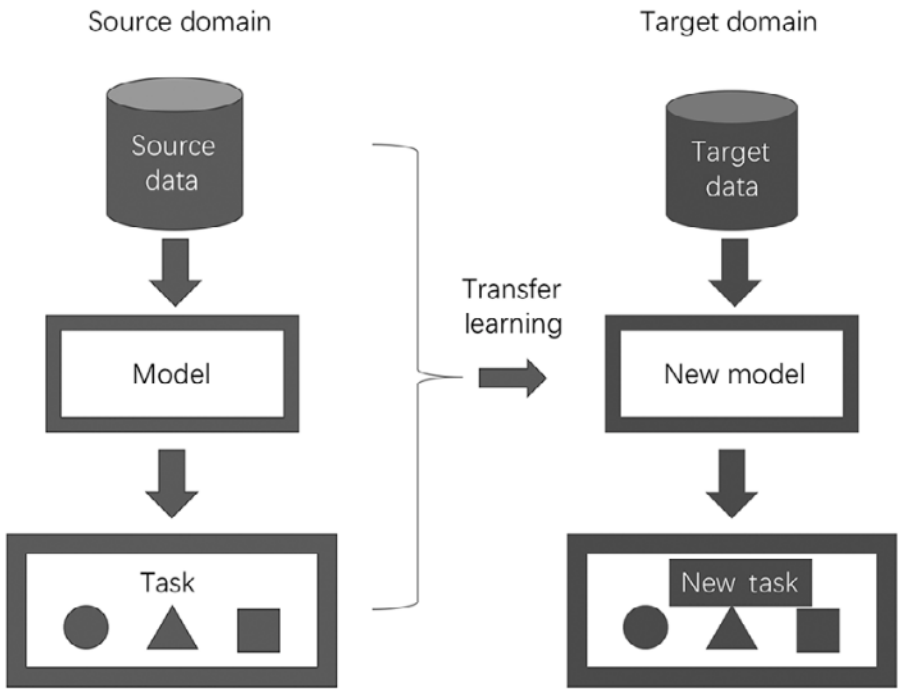

Figure 1.13: Illustration of a transfer learning process. Image source [41]

As a domain contains two components, $\mathcal{D} = \{\mathcal{X}, P^X\}$, the condition $\mathcal{D}_S \neq \mathcal{D}_T$ implies that either $\mathcal{X}_S \neq \mathcal{X}_T$ or $P^{X_S} \neq P^{X_T}$. Similarly, as a task is defined as a pair of components $\mathcal{T} = \{\mathcal{Y}, P^{Y|X}\}$, the condition $\mathcal{T}_S \neq \mathcal{T}_T$ implies that either $\mathcal{Y}_S \neq \mathcal{Y}_T$

or $P^{Y_S|X_S} \neq P^{Y_T|X_T}$. When the target domain and the source domain are the same , that is, $\mathcal{D}_S = \mathcal{D}_T$, and their learning tasks are the same, that is, $\mathcal{T}_S = \mathcal{T}_T$, the learning problem becomes a traditional machine learning problem.

Based on the homogeneity of the feature spaces and/or label spaces, transfer learning studies can be categorized into two settings: (1) homogeneous transfer learning and (2) heterogeneous transfer learning, defined by Pan [40] as it follows:

- **Homogeneous transfer learning**: *given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, heterogeneous transfer learning aims to help improve the learning of the target predictive function $f_{\mathcal{T}}(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{X}_S \cap \mathcal{X}_T \neq 0$ and $\mathcal{Y}_S = \mathcal{Y}_T$, but $P^{X_S} \neq P^{X_T}$ or $P^{Y_S|X_S} \neq P^{Y_T|X_T}$.*

- **Heterogeneous transfer learning**: *given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, homogeneous transfer learning aims to help improve the learning of the target predictive function $f_{\mathcal{T}}(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{X}_S \cap \mathcal{X}_T \neq 0$ or $\mathcal{Y}_S \neq \mathcal{Y}_T$.*

In the rest of this work the term “transfer learning" denotes homogeneous transfer learning.

Another categorization of transfer learning methods is given by the type of the traditional Machine Learning algorithms involved [42]:

- Inductive transfer: the source and the target domains are the same, while the source and the target tasks are different from each other. The algorithms try to utilize the inductive bias of the source domain to help improve the target task. Depending upon whether the source domain contains labeled data or not, this can be further divided into two subcategories, similar to multitask learning and self-taught learning, respectively.

- Unsupervised transfer: similar to the inductive transfer learning, with similar

domains and different tasks between source and target. In this scenario labeled data is unavailable in either of the domains.

- Transductive transfer: In this scenario, there are similarities between the source and target tasks but the corresponding domains are different. In this setting, the source domain has a lot of labeled data while the target domain has none. This can be further classified into subcategories, referring to settings where either the feature spaces are different or the marginal probabilities.

1.3.2 Algorithm Design

In order to design the transfer learning process, the following three questions must be answered:

- **When to transfer**: The first step in the whole process is to understand if the source domain and the target domain are related to each other. In situations where they are not, transferring knowledge may even hurt the performance of learning in the target domain, a situation which is often referred to as negative transfer.
- **What to transfer**: consists of identifying the portion of knowledge that is common between the source and the target, that may help improve the performance for the target domain or task.
- **How to transfer**: The last question to answer involves the identification of the methods to apply for transferring the knowledge across domains/tasks.

Based on the way the last question, *how to transfer*, is structured, four types of transfer learning algorithms can be identified:

- Instance-based algorithms: sometimes could be useful to directly transfer the training instances from the source to the target, eventually with some reweighting or resampling. This is motivated by the assumption that the source and the

target domains have a lot of overlapping features. In this context the knowledge transferred corresponds to the weights attached to source instances. However, in many real world applications, the assumption of overlapping features is too strong.

- Feature-based algorithms: this approach aims to learn a good feature representation for both the source domain and the target domain and then projecting the data onto a new representation where the source domain labeled data can be reused to train a precise classifier for the target domain. The knowledge transferred corresponds to the subspace spanned by the features in the source and target domains.

- Model-based algorithms: assuming that the source domain and the target domain share some parameters or hyperparameters of the learning models, first a deep learning model is trained on the source (large) data and then a few target labeled data are used to fine-tune part of the parameters of the pretrained deep model. So, the knowledge to be transferred is embedded in part of the source domain models.

- Relation-based algorithms: This approach is used when the relationships between objects are similar across domain or tasks and they not need to be independent and identically distributed (in contrast to the precedent methods). For instance, social network data utilizes relational-knowledge-transfer techniques [42]. The knowledge to be transferred corresponds to rules specifying the relations between the entities in the source domain.

1.3.3 Transfer Learning Strategies for Image Classification

Deep learning has made considerable progress in recent years and the results can be founds in the performances of SOTA models proposed on ImageNet Challenge, such as VGG, ResNet, Inception and so on. These pretrained models form the basis of

transfer learning in the context of deep learning and they have been applied to many fields other than image classification, like NLP, speech recognition, recommender systems, robotics and bioimaging [41].

The most-common approach for image classification is to transfer knowledge with model-based algorithms, taking care of the differences between the source and the target in terms of domain and available data.

Two strategies can applied in this scenario:

- **Pre-trained models as feature extractors**: When the target has few data available, pre-trained model's weighted layers are leveraged to extract features but not to update the weights of the model's layers during training with new data for the new task (this is obtained by *freezing* the layers). Usually, all the layers but the fully-connected ones are extracted from the source model and connected to a new fc layer.

- **Fine-tuning pre-trained models**: a more complex technique than just replace the final layer of the pre-trained model is to selectively retrain some of the previous layer. This approach can be useful when the target data are sufficient to carry some additional knowledge or the target labels are more plentiful. On the other side, using fine-tuning is more likely to result in overfitting.

1.3.4 Layer-by-Layer Transferability

Yosinski *et al.* [31] conducted various experiments in order to quantify the degree to which a particular layer is general or specific and how the trasfearability performs at different layers. In particular, they compared different combinations of transfer learning frameworks, based on the number of layers used for the feature extraction or fine-tuning. They observed that the performances of feature extractions is almost the same when using the first, the first two or the first three layers, meaning that not only the first layer is responsible of general features learning. Then the

performances degrades when using also the fourth and the fifth layers due to their *fragile co-adapted features*, that is, features that interact with each other in a complex or fragile way such that this co-adaptation could not be relearned by the upper layers alone. The drop of performances is more evident for the last layers, due to the conjoint effect of the lost co-adaptation and the representation specificity of the features.

The fine-tuning experiments instead showed an increase of performances when using more layers to fine-tune. The transferring features with fine-tuning provides a better generalization than training the network directly on the target dataset. Fine-tuning helps to recover co-adapted interactions between neurons.

Furthermore, they found that initializing with transferred features can improve generalization performance compared to initializing the network with random weights.

The fragile co-adaptation of neurons and the representation specificity of the last layers represent the main difficulties for a good trasfearability between networks.

1.3.5 Pre-trained Models

The pre-trained models used in computer vision are made available in the form of the millions parameters/weights the model achieved while being trained to a stable state on large datasets, such as ImageNet. In this Section we explore the architecture of one of the most popular model, VGG16.

1.3.6 VGG16

VGG16 is part of the Visual Geometry Group submission to the ILSVRC-2014. Simonyan and Zisserman [43] proposed 6 architectures with different depths (Figure 1.14) (ranging from 11 o 19 weight layers), but the same characteristics. In particular, they used very small 3x3 filters (which is the smallest size to capture the notion of left/right, up/down, center) in place of the 7x7 filters proposed by Krizhevsky's AlexNet. Although the combination of three 3x3 convolutional layers

(without spatial pooling in between) has the same receptive field of 7x7 filters, they found some improvements on reducing the receptive fields. First, the use of three non-linear activation layers instead of a single one makes the decision function more discriminative. Second, they substantially decreased the number of parameters.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 1.14: VGG configurations ordered for number of layers. The added layers for each configuration are shown in bold. Image source [43]

Two versions with 16 layers were proposed: one with 3x3 filters (configuration D, Figure 1.14) only and one with some 1x1 convolutions (configuration C, Figure 1.14). The incorporation of 1x1 convolutional layers is a way to increase the non-linearity of the decision function without affecting the receptive fields of the convolutional layer. This modification did not bring any improvement for the final

performance.

The use of stride $S = 1$ and padding $P = 1$ in the convolutional layers preserves the input dimensions. The downsampling is performed by Max-pooling layers (with receptive field 2x2, stride 2 and no zero-padding), placed at the end of each convolutional block. The input images have dimension 224x224 and the output width and height are halved at the end of each convolutional block according to the Max-pooling operations.

1.3.7 Discussions

Convolutional networks are able to learn themselves the features from the input. This is their strength and, for certain aspects, their weakness. The automatic feature learning is more robust than the former methods, such as SIFT, and have also outperformed these methods.

On the other hand, to learn meaningful features, ConvNets need very large dataset to train and the availability of such big dataset becomes a sort of competitive edge. Also, larger training dataset means larger training time and computational power required, that is not always available.

CNNs, like traditional machine learning algorithms, are designed to work in isolation: the training and test data are drawn from the same feature space and the same distribution. These algorithms works well unless the distribution changes. In this case most statistical models need to be rebuilt from scratch using newly collected training data.

Practically, data can be easily outdated, making the CNNs (and, more generally, machine learning algorithms) no more useful quickly. The need to build more robust models with respect to the training data and to not waste all the knowledge acquired during the training for a specific task has let come into play transfer learning.

Transfer learning exploited the basic idea of deep learning: first layers are responsible of low level features, that are common between different tasks. The

layer-by-layer trasfearability has been deepened and have led to several transfer learning schemes.

Transfer learning enabled the use of deep networks to new areas where the availability of images is limited and it can be seen as a democratization of deep learning, as computational resources and heavy datasets are no longer required to leverage the potential of deep learning.

Chapter 2

Hyperparameters Optimization

This chapter focuses on hyperparameter optimization (HPO), which aims to give a principled framework in order to choose the best model and its settings. The main difficulties faced by HPO are inherent within the multi-extremal and expensive function to be optimized in an agnostic search space. Most of the techniques proposed distinguish themselves by the way they walk in the search space, trading off exploration of less explored regions and exploitation of promising points. Two hyperparameters techniques, Bayesian optimization and Hyperband, are discussed in details, while their combination, BOHB, is deepened in the latest section of the chapter.

2.1 The role of hyperparameters

"FFN with a linear output layer and at least one hidden layer with any squashing activation function (such as the logistic sigmoid) can approximate any Borel measurable function from one finite-dimentional space to another with the desired

nonzero amount of error, provided that the network is given enough hidden units" [25].

Cybernenko (1989) and Hornink (1991) proved one of the first version of the universal approximation theorem, showing that any continuous function on a closed and bounded subset of R^n is Borel measurable and therefore may be approximated by a neural network.

Although in the original paper the theorem was first stated in terms of functions between two Euclidean spaces, many extensions have been proposed to deal with different scenarios, such as CNNs and radial basis functions [44]. Further considerations on universal approximation theorems are out of the scope of this work; the key point to focus on is that, regardless the function we are trying to learn, a large enough neural network will be able to represent it.

It's straightforward to point out that artificial neural networks can be used in a wide range of problems, making them the right tool to solve almost every learning problem.

AlexNet was the breakthrough, and at that time all the challenges that have held back Artificial Neural Networks for the past 50 years appeared to be surpassed: computational power is now stronger than ever (according to Moore's law the number of transistors in a dense integrated circuit doubles about every two years [45]) and massive amounts of data are available . So, what is the next challenge?

Although the universal approximation theorem states that a large enough neural network will be able to represent any function, it does not say anything about how large is enough nor about the feasibility of such ideal dimension and the capability of the net to learn and generalize correctly. Unfortunately, we have no prior information about how the choices in terms of width and depth of the model will impact the performances. The only evidence, as discussed by Goodfellow *et al.* [25] is that greater depth does seem to result in better generalization for a wide variety of tasks.

The importance of these architectural choices is marked by the "No free lunch theorem", (David Wolpert [46]), which states that an algorithm that scores well on one learning task can score poorly on another and even the same algorithm, with different hyperparameters values can show different performances.

The problem becomes bigger when we think about all the other prior decisions that affect learning. In fact, artificial neural networks require many other configurations to be set at prior: what optimizer, what learning rate, what activation functions, etc. .

All these design decisions are referred as neural networks hyperparameters and they represents a configuration that is external to the model and whose values cannot be estimated from data. Differently, the parameters are configuration variables that are internal to the model and whose values can be estimated from data.

The role of the hyperparameters becomes crucial for learning, because they guide the way how the model parameters are estimated.

2.2 Hyperparameter Optimization

The spread of deep neural network, complex, computationally expensive and characterized by many hyperparamaters resulted in the need of a research on the processes to choose the best set of hyperparamaters. This problem is known as hyperparameter optimization (HPO).

Although the rise of HPO frameworks coincided with the availability of powerful computational engines like graphical process units (GPUs), the first studies are dated back to the early 1990s [47] with the aim of outperform manual search and establish a fair framework for scientific studies, more reproducible than manual search.

The evaluation of a hyperparameter set needs to access the loss function of the model and the model itself has to be trained. This could result in very expensive

runs to evaluate different configurations on larger models. Furthermore, the hyperparameter space is a complex and multi-extremal black box, without derivative information available.

2.2.1 AutoMl

Hyperparameter optimization is only one of the tasks of Automated Machine Learning (AutoML). AutoML is a field that aims to automatize all the design decisions in a data-driven, objective, and automated way, in order to let the system determine the approach that performs best for a particular task. Often, AutoML has been switched with neural architecture search (NAS), the process of choosing the best neural architecture, but AutoML aspire to automatize many other decisions currently performed by humans, most of the time in a subjective way. In the most abstract perspective, the choice of an artificial neural network instead of a support vector machine is itself a decisions that could be automatized. This problem is known as *Full Model Selection* (FMS) or *Combined Algorithm Selection and hyperparameter optimization* (CASH).

Clearly, NAS, HPO and CASH are significant overlapping among them. The choice of the number of layers is an hyperparameter related to HPO, but a crucial choice for NAS, and also a conditioned hyperparameter of CASH when the selected algorithm is a neural network.

The methods applied in these subfields are often similar, with some adjustments for the specific task. In this section, the HPO problem is addressed, while more details about NAS and CASH can be found in [47].

2.2.2 Problem statement

Given a classification algorithm $\mathcal{A}$ with N hyperparameters λ_i defined in the overall configuration space Λ_N, let $\lambda \in \Lambda$ be a vector of hyperparameters while $\mathcal{A}$ with its hyperparameters is denoted by $\mathcal{A}_\lambda$. The problem of hypehyperparameter

optimizationrparameter space can be defined as it follows [47]:

$$\lambda^* = \arg\min_{\lambda \in \Lambda_N} \mathbf{V}(\mathcal{L}, \mathcal{A}_\lambda, D_{train}, D_{val}) \tag{2.1}$$

where $\mathbf{V}(\mathcal{L}, \mathcal{A}_\lambda, D_{train}, D_{val})$ measures the loss of a model generated by algorithm $\mathcal{A}$ with the hyperparameters set λ, trained on D_{train} and evaluated on D_{val}.

Most popular choices for validation protocol $\mathbf{V}(\cdot, \cdot, \cdot, \cdot)$ are Holdout, Iterated Holdout and Cross-Validation, which are applicable to user-defined loss functions.

The overall configuration space can contain continuous as well discrete hyperparameters and some of them can be conditional dependent on some others (e.g. the number of neurons in the third layer is relevant only if more than two layers exist). The conditionality can be represented by a tree structure, called *Direct Acyclic Graph*.

2.2.3 A global optimization problem

The goal of finding the configuration that results in the best performance measure defines the hyperoptimization as a global optimization problem.

Global optimization is a branch of applied mathematics and numerical analysis that attempts to find the global minima or maxima of a function or a set of functions on a given set [48]. A global optimization algorithm is guaranteed to find the global optimum if it runs long enough.

When applied to HPO, it is usually described as a minimization problem that aims to find the global minima of the performance metric (typically the loss function).

Since the loss function is non-convex and agnostic, in order to determine which is the best solution we have to compute the loss function for each possible configuration; clearly, it is an unfeasible brute-force approach. Usually, global optimization algorithms are designed to find the optimum with a given budget, that can be expressed in terms of time, memory, space, etc.

Most of the global optimization techniques are characterized by the way they "walk" through the unknown search space, hopefully discarding bad regions and focusing on prominent regions of this space. Given the complexity of the search space, each search strategy needs to address the trade-off between exploitation and exploration. The former tries to improve the best current solution sampling in promising regions of the space and it is associated to local search, while the latter diversifies the search sampling in agnostic regions, in order to avoid any local optimum trap and it is associated to global search.

The drawback is that this trade-off implies the introduction other hyperparameters, making the intervention of humans in the optimization loop necessary again (in the opposite way of the AutoML objective).

2.2.4 Black-box optimization

A wide range of mathematical function can be used when the performance metric $f(x)$ is treated like a black-box [47]:

$$\lambda^* = \arg\min_{\lambda \in \Lambda_N} f(x). \tag{2.2}$$

Methods in this category are independent from the model and due to the non-convex nature of the problem, global optimization algorithm are usually preferred. However, given the different nature of the hyperparameters (continuous, discrete and categorical), only derivative-free optimization methods can be applied.

In this section some of the most popular black-box methods are discussed.

Grid Serach

Grid search, also known as full factorial design, is the most basic method. Grid search needs a finite user-defined set of values for each hyperparameter and it evaluates the Cartesian product of these sets. In a simple scenario, with a few hyperparameters to be optimized and a cheap function f to evaluate, the grid search

can be useful as first-step method, in order to have an idea about the boundaries on each hyperparameter.

Grid search suffers of the curse of dimensionality as the number of function evaluations grows exponentially with the dimensionality of the configuration space. Grid search does not scale well, which makes it unsuitable for the context of deep learning, where the number of hyperparameters is too large and each function evaluation is expensive.

An additional problem of Grid search is that it always tests the same limited set of values for each hyperparameter.

Random Search

Grid search treats equally the different hyperparameters. In deep learning context, some (few) hyperparameters have an higher impact on the performances than others [49].

Random Search takes advantage from this assumption and draws the value of each hyper-parameter from a uniform distribution, allowing for a much wider range of explored values. Actually, when compared to grid search in a hyperparameter space of two hyperparameters, one more important than the other, random search results in a better solution, as shown in Figure 2.1.

Given the independence of each configuration, random search is embarassingly parallelizable.

While the computational cost is the same of grid search, random search is preferred for its results, better parallelization and flexible resource allocation. This makes such method useful as baseline or initialization of the search process, as it explores the entire configuration space and gives good information about the hyperparameters boundaries (better than grid search).

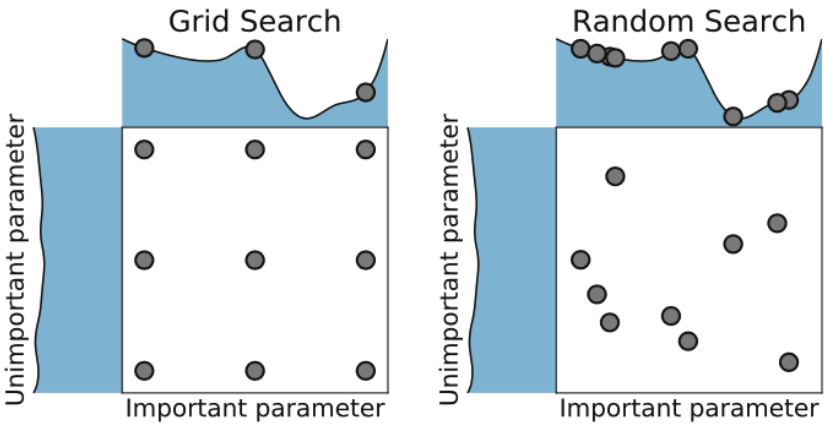

Figure 2.1: Comparison of Grid Search (left) and Random Search (right) for minimizing a function with one important and one unimportant parameter. Grid search evaluates only 3 values for each hyperparameter, while Random search tests 9 different configurations for each method, resulting in a better solution. Image source [47]

Bayesian optimization

Bayesian Optimization (BO) is a state-of-the-art optimization (SOTA) framework for the global optimization of expensive black-box functions [47] that gained SOTA results in deep learning models for image classification, speech recognition and natural language processing.

It is based on the assumption that similar values of hyperparameters result in similar performances (a weak form of continuity). At each step, BO determines which is the best point of the hyperparameter space to evaluate through its activation functions. At the same time, a model describing estimates and uncertainty at each point is updated according to the Bayes' rule. Bayesian optimization balances exploring uncertain regions, which might unexpectedly give good performances (exploration), against focusing on regions that already showed good performances (exploitation).

Many recent developments in BO do not treat HPO as a black-box anymore, combining it with multi-fidelity methods [50] or meta-learning[51].

Bayesian optimization is explained in details in Section 2.3 .

Other approaches

The most common approach in solving global optimization problems (not only HPO) when only weak, or even no one, assumptions can be made, is the use of metaheuristics. They are problem-independent techniques that guide the search process with incorporated stochastic mechanisms that avoid getting trapped in local minima.

Several metaheuristcs have been deployed in hyperparameter optimization, like genetic algorithm [52], simulated annealing [53] or tabu search [54].

The most recent advances in HPO have been addressed with the use of reinforcement learning pipelines, i.e. training an agent able to design neural networks for specific tasks [55].

2.2.5 Multi-fidelity optimization

Along with the need of evaluating several configurations, current methods must be able to deal with the always increasing size of datasets, which training could require several hours or days and very powerful (and expensive) hardware. For example, GPT-3, the state-of-the-art model for natural language processing (NLP), is trained on 499 billion tokens, has 175 billion parameters and his training costs is estimated around $4,600,000 even with the lowest priced GPU cloud on the market [56].

Although optimizing GPT-3 is useless, it is clear how datasets and nets are growing and they will continue to (at least the datasets will).

Multi-fidelity algorithms use low fidelity approximations of the actual loss function to minimize; this approximations are obtained by training the network on subsets of the datasets and/or each configuration only for a few iterations and/or by using downsampled data (e.g. downsampled images for computer vision) and/or by using only a subset of features, ...

Successive Halving and Hyperband

Successive Halving (Jamieson and Talwalkar, 2015 [57]) is one of the most used bandit-based algorithm selection methods. It exploits a simple but powerful concept: given an initial budget, it tests all the proposed configurations with the same budget, then remove the half performed worst, doubles the budget and successively repeats until only a single configuration is left [47].

While its results are robust, it requires the user to make explicit the trade-off between exploration and exploitation a priori. In fact, the user must first decide whether to assign many budget on few configurations or lower budget to many different configurations.

To overcome this problem, another techinque called Hyperband is deployed. It divides the total budget into several combinations of budget vs configurations and then calls successive halving as a subroutine. Hyperband is detailed in Section 2.5 .

The bandit-based algorithm selection methods are extremely simple and they can be viewed as formalized metaheuristics. Although their performances as standalone methods are satisfying, they achieve SOTA results when combined with other methods that can adapt the configuration proposal strategy to the past function evaluations.

The combination of Hyperband with Bayesian optimization, i.e. BOHB, is one of the most interesting one, and it is discussed in Section 2.6.1.

2.2.6 Optimization methods desired characteristics

Practical applications of hyperoptimization algorithms face complex scenarios, where as many as possible hyperparameters need to be set respecting some computational constraints.

Hutter *et al.* [58] listed the desiderata that any HPO method should fulfill:

- **Strong Anytime Performance.** The computational complexity of most of the functions to be optimized results in extremely large training times. Often the available budget for the optimization is not much more than that of a single training, so practical HPO methods should already yield good configurations with small budget.

- **Strong Final Performance.** Although the need to obtain good results with small budget, what matters most is the best configuration an HPO method can find using all the available budget. The final performance needs to go next to the global optimum.

- **Effective Use of Parallel Resources.** To exploit the power of parallel resources, any practical HPO method needs to be able to use these effectively.

- **Scalability.** Following the aim of AutoML, i.e. automatize all the prior design choices, HPO methods must be able to easily handle problems ranging from just a few to many dozens of hyperparameters.

- **Robustness & Flexibility.** The HPO methods must deal with different types of hyperparameters (e.g. conditional, discrete and continuous) and should apply in several domains of machine learning.

Almost all existing methods have some strengths and weakness. We describe more in details an *effective* method, Bayesian optimization, and an *efficient* method, Hyperband. Later, their combination is analyzed and evaluated with respect to the above desiderata.

2.3 Bayesian Optimization

Bayesian optimization is an iterative algorithm composed of two main ingredients. The first component is a surrogate model of the objective function that uses the

hyperparameters' values as inputs and gives an estimate of the loss function associated to the model (e.g. a neural network). Once obtained information about the loss (and its uncertainty) in the hyperparameter space, the second component, called activation function, decides which point to evaluate next.

2.3.1 Gaussian Processes

A Gaussian Process (GP) is a supervised learning model that allow us to make predictions about data by incorporating prior knowledge. It is mainly used for fitting a function to the data (i.e. regression), but it can be extended to other tasks, such as classification and clustering.

The Gaussian distribution is the cornerstone of GPs. While a Gaussian distribution is a distribution over vectors, a Gaussian Process is a distribution over function.

First, the Gaussian distributions are introduced, then we'll focus on GPs and its hyperparameter.

Multivariate Gaussian Distribution

A Gaussian (or Normal) distribution is a type of continuous probability distribution for real-valued random variables. The general form of its probability density function is [59]:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{2.3}$$

where the parameter μ is the mean or expectation of the distribution (and also its mode and median) and σ is its standard deviation. A random variable with Gaussian distribution is said to be normally distributed.

A Multivariate Gaussian (or Normal) distribution is its generalization, from a one-dimensional space to higher dimensions. One definition is that a random

vector is said to be k-variate normally distributed if every linear combination of its k components has a univariate normal distribution [60].

While the univariate Gaussian distribution is defined by mean μ and standard deviation σ, the multivariate one is defined by a mean vector μ and a covariance matrix Σ. The first describes the expected value of the distribution and each of its components describes the mean of the corresponding dimension. The covariance matrix Σ, always symmetric and positive semi-definite, models the variance along each dimension and determines how the different random variables are correlated.

Among the many properties that characterize Gaussian distributions, marginalization and conditioning [61] are the most important for the objective of this study.

Both these properties work on subsets of the original distribution. Given the multivariate probability distribution:

$$P_{X,Z} = \begin{bmatrix} X \\ Z \end{bmatrix} \sim \mathcal{N}(\mu, \Sigma) = \mathcal{N}\left(\begin{bmatrix} \mu_X \\ \mu_Z \end{bmatrix}, \begin{bmatrix} \Sigma_{XX} & \Sigma_{XZ} \\ \Sigma_{ZX} & \Sigma_{ZZ} \end{bmatrix} \right) \tag{2.4}$$

where X and Z are subsets of the original random variables, the **marginalization** allows to extract partial information (i.e. the distribution of the subsets X and Z) from a multivariable probability distribution without referencing to the values of other variables. In particular, the marginalized probability distributions of X and Z are defined in the following way:

$$X \sim \mathcal{N}(\mu_X, \Sigma_{XX}) \tag{2.5a}$$

$$Z \sim \mathcal{N}(\mu_Z, \Sigma_{ZZ}). \tag{2.5b}$$

In this way, each partition X and Z only depends on its corresponding entries in μ and Σ.

In contrast with marginality, the property of **conditioning** is used to determine the probability of one variable depending on another variable. The conditional marginal distributions are defined by:

$$X|Z \sim \mathcal{N}(\mu_X + \Sigma_{XZ}\Sigma_{ZZ}^{-1}(Z - \mu_Z), \Sigma_{XX} - \Sigma_{XZ}\Sigma_{ZZ}^{-1}\Sigma_{ZX}) \tag{2.6a}$$

$$Z|X \sim \mathcal{N}(\mu_Z + \Sigma_{ZX}\Sigma_{XX}^{-1}(X - \mu_X), \Sigma_{ZZ} - \Sigma_{ZX}\Sigma_{XX}^{-1}\Sigma_{XZ}). \tag{2.6b}$$

The mean of each conditional distribution only depends on the conditioned variable, while the covariance matrix is independent from this variable.

Definitions

A Gaussian Process is a collection of random variables, any finite number of which have a joint Gaussian distribution [62].

A Gaussian Process is completed defined by its mean function and covariance function. The mean function $m(x)$ and the covariance function $k(x, x')$ of a real process $f(x)$ are defined as:

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)] \\ k(x, x') &= \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))] \end{aligned} \tag{2.7}$$

and the underlying Gaussian Process:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')) \tag{2.8}$$

The mean is often assumed to be $m(x) = 0$, which simplifies the equations for conditioning. In practice this is ensured by removing the mean of the predicted values from the dataset. However, it can be added back to the resulting function values after the prediction step (in a process called centering of the data).

The covariance function between two points returns a similarity measure between those points in the form of a scalar. All the pairwise evaluations are used to build the Covariance matrix:

$$K(X,X) = \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \dots & k(x_1,x_N) \\ k(x_2,x_1) & k(x_2,x_2) & \dots & k(x_2,x_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_N,x_1) & k(x_N,x_2) & \dots & k(x_N,x_N) \end{bmatrix} \tag{2.9}$$

Each entry Σ_{ij} desribes how much influence the i-th and j-th components have on each other.

Since the kernel describes the similarity between the values of the function, it controls the possible shape that a fitted function can adopt.

The marginalization property is a consistency requirement for Gaussian Processes, that is automatically fulfilled if the covariance function specifies entries of the covariance matrix. This property ensures that examination of a larger set of variables does not change the distribution of the smaller set.

Prior Distribution

The specification of the covariance function implies a distribution over functions. We can pick a set of input points (test points), X_*, build the covariance matrix, $K(X_*, X_*)$, and generate samples from this Gaussian distribution:

$$f_* \sim \mathcal{N}(0, K(X_*, X_*)) \tag{2.10}$$

Figure 2.2 shows some of the samples taken from (20). Since no training data has been used, (10) corresponds to an unfitted Gaussian Process, called Prior distribution in the context of Bayesian inference.

The prior distribution has the same dimensionality as the number of test points $N = |X_*|$, while the covariance matrix has dimension $N \times N$.

As the prior distribution does not contain any additional information about the function, it clearly shows the influence of the kernel on the distribution of functions.

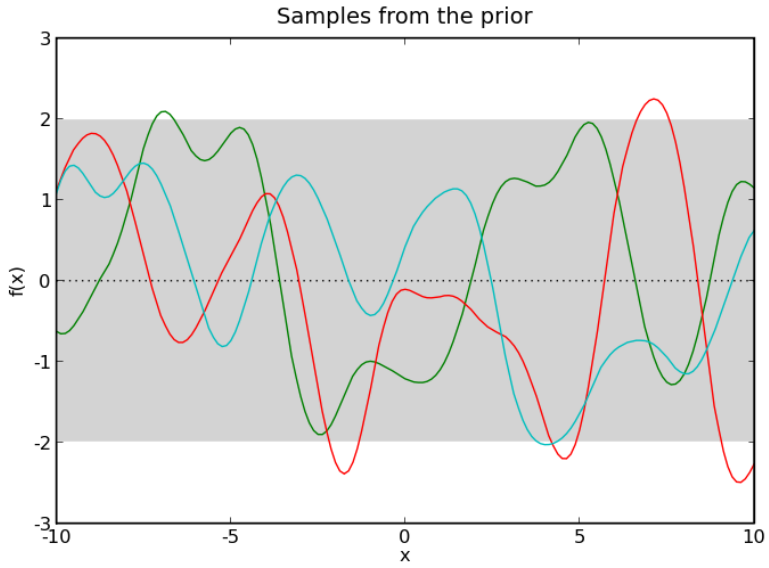

Figure 2.2: Three samples drawn at random from a GP prior. The gray area is the 95% prediction interval around the mean. Image source [63]

Kernels

As mentioned above, the covariance function is the keypoint of a Gaussian Process, as it encodes our assumptions about the function which we want to learn. The term *kernel* derives from therory of integral operators and represents a general function k of two arguments mapping a pair of inputs $x \in X, x' \in \mathcal{X}$ into $\mathbb{R}$. A kernel is said to be symmetric if $k(x, x') = k(x', x)$ and covariance functions are symmetrical by definition.

Several kernels have been proposed and adopted in the statistical literature; several studies focused both on the right choice of the kernel for each specific problem and the combination of different kernels.

The most common is the squared exponential kernel:

$$k(x, x') = \sigma^2 exp\left(-\frac{||x - x'||_2^2}{2l^2}\right) \tag{2.11}$$

where σ^2 is the *signal variance* and determines variation of function values from

their mean, while l is the *characteristic length-scale*, that determines how smooth the function is.

The use of this kernel causes decay of the influence of a point on the value of another point exponentially with their relative distance. This implies that the Gaussian Process reverts to its priors in not observed areas.

The samples resulting from the squared exponential kernel are very smooth because it is infinitely differentiable. Stein (1999) argues that such strong smoothness assumptions are unrealistic for modelling many physical processes.

A more realistic alternative is represented by the Matérn 5/2 kernel, a special kernel from the Matérn class:

$$k(r) = \sigma^2\left(1 + \frac{\sqrt{5}r}{l} + \frac{5r^2}{3l^2}\right)\exp\left(-\frac{\sqrt{5}}{l}\right) \tag{2.12}$$

which produces samples twice differentiable, that is a good compromise between computational difficulty and smoothness.

Both these covariance functions are stationary, i.e. are invariant to translations in the input space and depend only on the difference $x - x'$. This means that in the context of HPO the kernels make no difference between values of 1 and 2 or 99 and 100.

For some situations where an hyperparameter is sensitive to the scale, non-stationary kernels, like the dot-product ones, are preferred.

Posterior Distribution

Although the prior itself gives some information about the function to approximate, the primary objective of Gaussian Processes is to incorporate the knowledge provided by some observed points, i.e. the training data.

Given a set of observed points $\{(X_i, f_i)|i = 1, \ldots, n\}$ and a set of test points we want to predict $\{(X_{*i}, f_{*i})|i = 1, \ldots, n_*\}$, the joint distribution of the training outputs, f, and the test outputs, f_*, according to the prior in (10) is

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*)). \end{bmatrix}\right) \tag{2.13}$$

The matrix $K(X, X_*)$ denotes the $n \times n_*$ matrix of covariances evaluated at all pairs of training and test points. The other matrices $K(X, X)$, $K(X_*, X)$ and $K(X_*, X_*)$ are defined in a similar way.

Similarly to the prior distribution, the prediction of the posterior distribution could be obtained by sampling from (2.13). However, to get meaningful predictions we need to restrict the joint prior distribution to contain only those functions which agree with the observed data points. This is obtained by conditioning the joint prior distribution to the observations, so as to give:

$$f_*|X_*, X, f \sim \mathcal{N}(K(X_*,X)K(X,X)^{-1}f, \\ K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*)). \tag{2.14}$$

Figure 2.3 shows on the left some samples obtained before using training points (i.e. prior distribution) and on the right some samples obtained yielded by the posterior distribution.

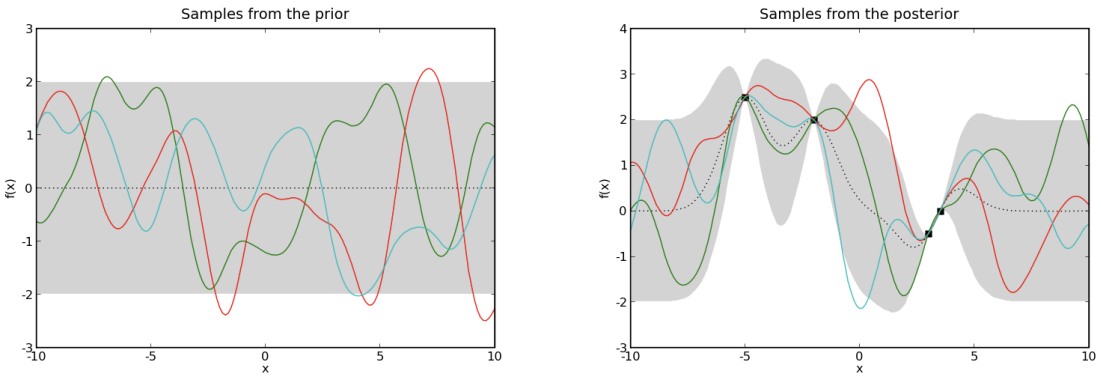

Figure 2.3: Comparison of prior distribution (left) and posterior distribution (right) on a one dimensional problem. The 4 observed points in the posterior distribution are denoted by squares. Image source [63]

The training points considered above are assumed to be perfect measurements. For more realistic situations, we do not have access to the function values themselves, but rather we can use noisy versions $y = f(x) + \eta$, where η independent identically distributed Gaussian noise with variance σ_n^2. The prior on noisy observations becomes:

$$y \sim \mathcal{N}(0, K(X, X) + \sigma_n^2 I) \tag{2.15}$$

Introducing the diagonal matrix $\sigma_n^2 I$ to the noise free case in (2.13), we can write the joint distribution with noisy observations as it follows:

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X, X) + \sigma_n^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*)) \end{bmatrix}\right) \tag{2.16}$$

Again, we use conditioning to derive the predictive distribution $P_{X|y}$ from (16).

2.3.2 Acquisition Functions

The Gaussian Process provides an estimation of the performances associated to each set of hyperparameters and its uncertainty. Then, the acquisition function is deployed to choose which configuration to evaluate next, trading off exploration and exploitation. Compared to evaluating the expensive blackbox functions, the acquisition function is cheaper to compute and can therefore be thoroughly optimized.

Typically, acquisition functions are defined such that high values correspond to potentially higher values of the objective function, whether because the prediction is high, or the uncertainty is great, or both.

In this section two popular acquisition functions are presented, Probability of Improvement and Expected Improvement. Many other acquisition functions could be found in [64, 65].

Probability of Improvement

One of the oldest acquisition functions was proposed by Kushner [66] under the name of Probability of Improvement (PI). It chooses the configuration which has the maximum probability of having better results than the best observed so far. The PI is defined as it follows:

$$PI(x) = \Phi\left(\frac{\mu(x) - \mu^{+} - \xi}{\sigma(x)}\right) \tag{2.17}$$

where $\Phi(\cdot)$ is the Normal cumulative distribution function, μ^{+} is the best configuration found so far (incumbent), $\mu(x)$ is the mean returned by the Gaussian Process and $\sigma(x)$ its variance.

This acquisition function is biased by definition toward exploitation only, because it will sample configurations very close to the corresponding model. To remedy this, the trade-off parameter $\xi \geq 0$ is used. Higher values of this parameter drive to exploration, while lower values drive to exploitation. Kushmer recommends to use a schedule for ξ, which should start high and then decrease to zero as the algorithm progresses.

Expected Improvement

The PI acquisition function does not take into account how much the function would improve, but only the probability of the improvement.

Mockus *et al.* [67] proposed the Expected Improvement (EI) that focuses on the magnitude of the improvement a point can potentially yield. The EI acquisition function is obtained as:

$$EI(x) = \begin{cases} d\Phi(d/\sigma(x)) + \sigma(x)\phi(d/\sigma(x)) & \text{if} \sigma(x) > 0 \\ 0 & \text{if} \sigma(x) = 0 \end{cases} \tag{2.18}$$

where $d = \mu(x) - \mu^{+} - \xi$ and where $\phi(\cdot)$ and $\Phi(\cdot)$ denotes the probability

distribution function (PDF) and the cumulative distribution function (CDF) of the standard Normal distribution, respectively.

2.3.3 Bayesian Optimization Algorithm

Once defined the main components of the Bayesian optimization, the probabilistic surrogate model and the activation functions, we can combine them to obtain the optimization algorithm. It is important to note that Gaussian Process, although commonly, it is only one of the possible surrogate models for BO. Random forests are a valid alternative, as they handle conditional hyperparameters better than GPs do [47].

Furthermore, the available activation functions are many more than the two presented in Section 3.2 and can be tailor made for specific problems.

A full exploration of possible surrogate models and activation functions is out of the scope of this thesis, but whatever the choice, the pipeline of Bayesian optimization remains the same.

Given a set of observation $D_{1:t} = \{x_{1:t}, y_{1:t}\}$ the surogate model (e.g. Gaussian Process) is fitted and returns an estimate of the performance of the hyperparameters' values. Then the acquisition function chooses the configuration to evaluate next. This procedure, depicted in Figure 2.4 is iterated until the optimum is found or, more likely, when all the available resources are consumed.

Algorithm 1: Bayesian Optimization Algorithm

1 **for** *t=1,2, ...* **do**
2 Find the configuration x_t by maximizing the activation function α over the surrogate model f ;
3 Sample the objective function $y_t = f(x_t) + \epsilon$;
4 Augment the data $D_{1:t} = D_{1:t-1}(x_t, y_t)$ and update the posterior of function f
5 **end**

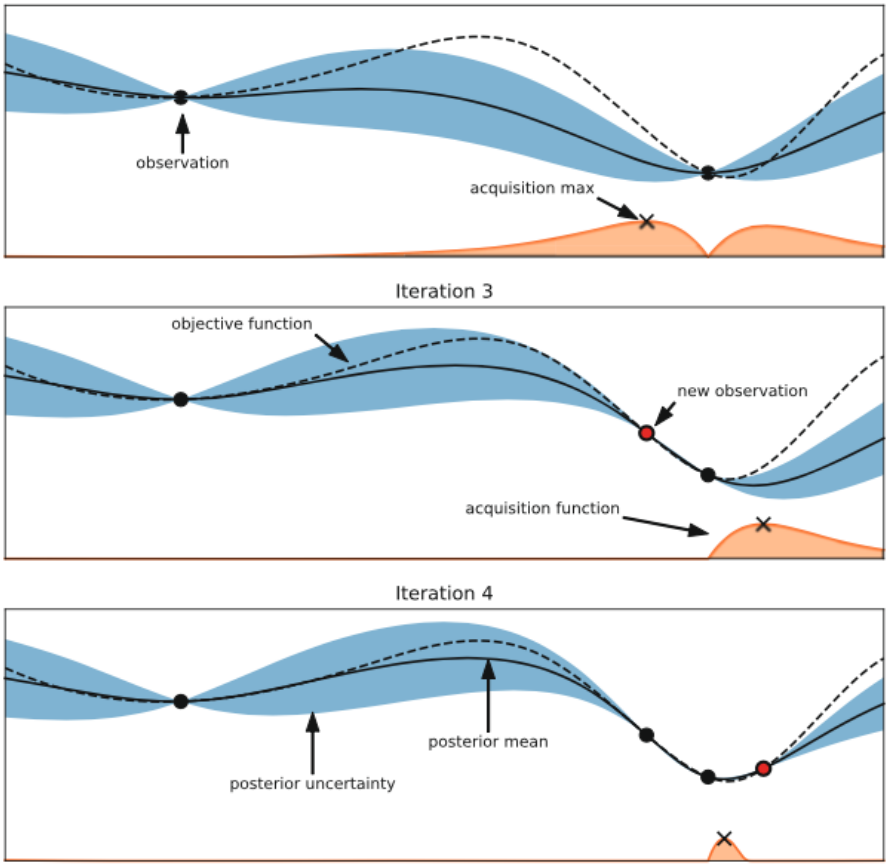

Figure 2.4: Illustration of Bayesian optimization on a 1-d function. The goal is to minimize the dashed line using a GP by maximizing the acquisition function represented by the lower orange curve. Image source [47]

The main drawback of Bayesian optimization lies in the computational costs of the surrogate model, due to the construction of the Gram Matrix K and the computation of its invert K^{-1}. To overcome that cost, often it is used the incremental Cholesky decomposition, that leads to a lower computational complexity, that is $O(n^3)$, for fitting the GP.

2.3.4 Tree Parzen Estimator

Although the incremental Cholesky decomposition reduces the computational effort, it is still a bottleneck for Gaussian Processes.

An alternative approach is to use the Tree Parzen Estimator (TPE) [68], which instead of modelling $p(y|x)$ directly, takes advantage from the Bayes' rule and models $p(x|y)$ and $p(y)$ separately.

The TPE replaces the distribution of the configuration prior with non-parametric densities. By using different observations $\{x_1, \ldots, x_k\}$ in the non parametric densities, these substitutions represent a learning algorithm that can produce a variety of densities over the configuration space Λ. The TPE defines $p(x|y)$ by using the following densities:

$$p(x|y) = \begin{cases} l(x) & \text{if} y < y^* \\ g(x) & \text{if} y \geq y^*, \end{cases} \tag{2.19}$$

where $l(x)$ is the density function of the observations $\{x_i\}$ such that the corresponding loss $f(x_i)$ is less than y^* and $g(x)$ is the density obtained by using the remaining observations. Whereas the GP-based approach favours quite an aggressive y^+, the TPE algorithm depends on a y^+ that is larger than the best observed $f(x)$, so that some points can be used to form $l(x)$. The TPE algorithm chooses y^+ to be some quantile γ of the observed y values, so that $p(y < y^+) = \gamma$, but no specific model for $p(y)$ is necessary.

The parametrization of $p(x, y)$ as $p(y)p(x|y)$ in the TPE algorithm is chosen to facilitate the optimization of EI. Recalling from (2.18):

$$EI(x) = \int_{-\infty}^{y^+} (y^+ - y)p(y|x)\, dy = \int_{-\infty}^{y^*} (y^+ - y)\frac{p(x|y)p(y)}{p(x)} dy \tag{2.20}$$

By construction, $\gamma = p(y < y^+)$ and $p(x) = \int_{\mathcal{R}} p(x|y)p(y)dy = \gamma l(x) + (1 - \gamma)g(x)$. Therefore

$$EI(x) = \int_{-\infty}^{y^+} (y^+ - y)p(y|x)\, dy = l(x) \int_{-\infty}^{y^+} (y^+ - y)p(y)\, dy = \gamma y^+ l(x) - l(x) \int_{-\infty}^{y^+} p(y)\, dy \tag{2.21}$$

So that finally:

$$EI(x) = \frac{\gamma y^{+} l(x) - l(x) \int_{-\infty}^{y^{+}} p(y)\, dy}{\gamma l(x) + (1-\gamma) g(x)} \propto \left(\gamma + \frac{g(x)}{l(x)}(1-\gamma)\right)^{-1} \tag{2.22}$$

To select a new candidate to evaluate, we need to maximize the ratio $l(x)/g(x)$ that is, according to Bergstra *et al.* (2011) [68], equivalent to maximize EI in (18). The tree-structured form of l and g makes it easy to model mixed continuous and discrete spaces, and the model construction scales linearly in the number of data points (that is a big gain with respect to the cubic time of Gaussian Processes).

2.4 Hyperband

Hyperband [69] extends the successive halving algorithm presented in Section 2.2.5 and calls it as a subroutine.

The problem of successive halving is that it makes explicit the trade-off between exploration and exploitation.

More specifically, given some finite budget B (e.g., memory, time, epochs) and n configurations specified by the user, B/n resources are allocated on average across the configurations. However, for a fixed budget, it is not clear a priori whether to chose a large n, considering many configurations with a small average training time, or considering a small number of configurations (small n) with longer average training times.

Hyperband, as explained by Giulia DeSalvo *et al.* (2018), addresses the "n versus B/n" problem by considering several possible values of n for a fixed B. Some minimum resources, r, are allocated to all n configurations before some are discarded by successive halving. Hyperband, as shown in Algorithm 2, is made of two components: (1) the inner loop that invokes successive halving for fixed values of r and n (each run of such loop is called "bracket") and (2) an outer loop that iterates over different values of r and n, alternating some "exploratory"

successive halving runs with some more “exploitative" ones, using approximately B total resources.

Algorithm 2: Hyperband algorithm using Successive Halving as subroutine

Input : R, η

Initialization: $s_{max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{max} + 1)R$

s $\in \{s_{max}, s_{max-1}, \ldots, 0\}$n $= \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil$, $r = R\eta^{-s}$

// begin SuccessiveHalving with (n, r) inner loop

T = sample n configurations

for $i \in \{0, \ldots, s\}$ **do**

  $n_i = \lfloor n\eta^{-i} \rfloor$

  $r_i = r\eta^i$

  L = validation loss $\{(t, r_i) : t \in T\}$

  $T = \text{top}_k$(T, L, $\lfloor n_i/\eta \rfloor$)

Return : Configuration with the smallest validation loss so far

Hyperband performs a geometric search in the average budget per configuration and removes the need to select n for a fixed budget. It requires only two inputs: the maximum amount of resources that can be allocated to a single configuration, R, and η, a parameter that controls how many configurations are discarded in each run of a bracket. These two inputs combined control the number of bracket considered; specifically, $s_{max} + 1$ different values for n are considered with $s_{max} = [\log_\eta(R)]$. Hyperband starts with $s = s_{max}$, which sets n to maximize exploration, subject to the constraint that at least R resources are allocated to one configuration. Each subsequent bracket reduces n by a factor of approximately η until the final bracket, $s = 0$, in which every configuration is allocated the maximum resources R.

Hyperband starts to make progress much faster (20x) than random search and gives a 4x speed-up for final performances (Figure 2.5).

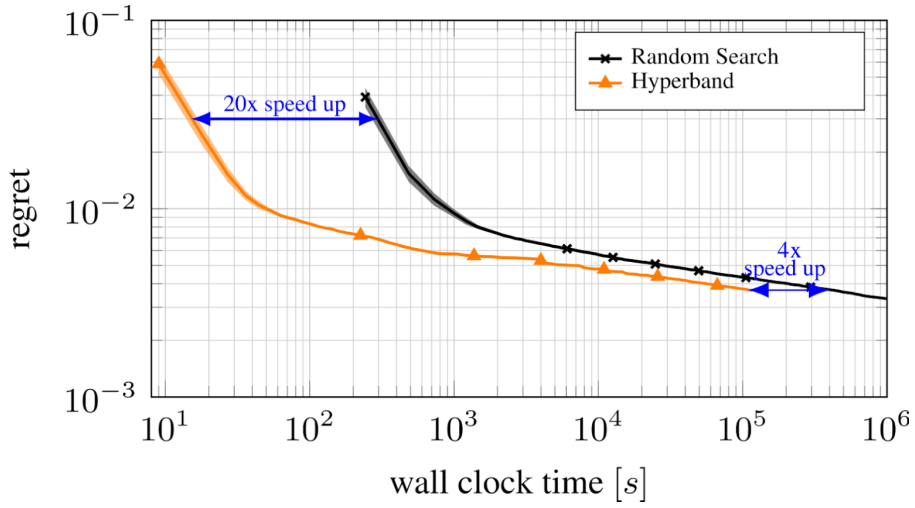

Figure 2.5: Comparison of performances progression between Hyperband and Random Search. Image source [70]

2.5 Bayesian Optimization and Hyperband

Bayesian optimization gives the SOTA final performance but does not scale well due to its cubic complexity in the number of data points, does not fit to complex configuration spaces without special kernels (flexibility) and requires carefully-set hyperpriors (robustness).

On the other hand, Hyperband outperforms BO for anytime performance, but its randomic nature results in a slower finding of the best configuration (final performance).

In order to fulfill all the desiderata presented in Section 2.5, Falkner at al. (2017) [71] proposed to combine Bayesian optimization and Hyperband in a new hyperoptimization approach.

Later studies have led to the formalization of BOHB (Falkner *et al.*, 2018) [58], a robust optimization framework that takes advantage of the strong final performance from BO and of the strong anytime performance, flexibility, scalability and parallelization from HB.

2.5.1 BOHB

BOHB relies on HB to determine how many configurations to evaluate with which budget; it replaces the HB random selection of configurations at the beginning of each HP run by a model-based search, using TPE as a BO component. Once the desired number of configurations for the iteration is reached, the standard successive halving procedure is carried out using these configurations, and the performance across all budgets are recorded to use them as a basis for the models in later iterations.

The substitution of the random sampling with a model-based sampling allows to use the information of the previous evaluated configurations and it leads to a better search of the global optimum, by keeping the advantages of Hyperband.

Rather than using the hierarchy of one-dimensional kernel density estimators (KDEs), typical of TPE, a single multidimensional KDE, which constructs a factorized kernel with one-dimensional kernel for each dimension is preferred, in order to better handle interaction effects in the input space.

For a useful KDE, a minimum number of configurations N_{min} is required (usually it set to $d+1$ or $d \cdot 2$ [71], where d is the number of hyperparameters.) To build a model as soon as possible, we do not wait until $N_b = |D_b|$, but it is sufficient that the number of observations for budget b is large enough to satisfy $q \cdot N_b \geq N_{min}$. Indeed, after initializing with $N_{min} + 2$ random configurations, we choose the

$$\begin{aligned} N_{b,l} &= \max(N_{min}, q \cdot N_b) \\ N_{b,g} &= \max(N_{min}, N_b \cdot N_{b,l}) \end{aligned} \tag{2.23}$$

best and worst configurations, respectively, to model the two densities in (19). This ensures that both models have enough data points and have the least overlap when only a limited number of observations is available [58].

Sometimes, the lower fidelities can be misleading, i.e. one algorithm that is

slower to converge to the solution can give poor anytime performance, but at the same time it can reach the best final performance. To overcome the problem of making wrong conclusions drawn from a smaller budget, more configurations are evaluated on bigger budgets when the optimization progresses. Once the two densities $l(x)$ and $g(x)$ form (19) are learned, N_s samples are drawn from $l'(x)$, that is the same as $l(x)$, but with all bandwidths multiplied by a factor b_w to encourage more exploration around the promising configurations. Finally, the configurations with the highest ratio $l(x)/g(x)$ are selected for the next evaluation.

In order to keep the explorative randomic nature of HB, at each run of SH a constant fraction ρ of random configurations is evaluated along the model-based picks. A full-view of BOHB sampling procedure is detailed in Algorithm (3).

Algorithm 3: BOHB sampling pseudocode

Input : Observations D, fraction of random runs ρ, percentile q, number of samples N_s, minimum number of points N_{min} to build a model, bandwidth factor b_w

Output: next configuration to evaluate

If: $rand() < \rho$ **then return** random configuration

$b = \arg\max\{D_b : |D_b| \geq N_{min} + 2\}$

If: $b =$ ‰ **then return** random configuration

fit KDEs according to Eqs. (19) and (23)

draw N_s samples according to $l'(x)$

Return: sample with highest ratio $l(x)/g(x)$.

No optimizer is free from hyperparameters itself, but the studies conducted by Falkner *et al.* [58] showed that BOHB is quite insensitive to its hyperparameters, with the default values working robustly across different scenarios. The only challenging hyperparameter is b_{min}, which wrong initialization could result in poor optimization. While the maximum budget is usually known and derived by the available resources, the minimum budget needs to be small to benefit the multi-fidelities approach, but large enough to give robust information about the model

performances. An accurate setting of b_{min} requires knowledge about the benchmark and the algorithm being optimized.

2.6 Discussions

The hyperparameters play a crucial role in the model effectiveness since they determine the convergence of the model to the optimum. The relevance of HPO matches with the difficulty of its implementation. No prior information is available to address the trade-off between exploration and exploitation and as the number of hyperaparameters grows (along with their heterogeneity), the search space become more complex.

Although the random-based methods such as random search and Hyperband result very efficient in testing many configurations, they do not learn anything from the previous evaluations, making the search dummy and not suitable for the global optimum search. They are mainly used as benchmarks thanks to the effective use of parallel resource, scalability, robustness and flexibility.

In the other side the effectiveness of the Bayesian optimization is strongly affected from its computational cost and from the nature of the hyperparameters. The replacement of the Gaussian process with the TPE makes the Bayesian optimization suitable also for categorical hyperparameters and reduces the computations required, but not as much as needed to explore enough the search space.

BOHB is designed in order to take advantage of the knowledge acquired by Bayesian optimization during the search process and from the speed-up given by the low-fidelities approximations. Also, the Hyperband randomic nature helps to mitigate the exploitative nature of Bayesian optimization.

Chapter 3

Damage Detector

The strong improvements in machine learning techniques and the developments in technology allowed to apply these paradigms to many business applications. This chapter focuses on the application of BOHB hyperoptimization on convolutional neural networks, in order to develop a new solution for insurance market: automatic detection of car damages. The hyperoptimization is first conducted on a CNN build from scratch and then on the fine-tuning of a pre-trained model. Once the optimal model is found, it is deployed in a web application.

3.1 Project overview

3.1.1 Business perspective

Most online insurance companies can't verify the existence and condition of the vehicle during policy application, and this lead to fraudulent behaviours (e.g., purchasing a policy for a previously damaged vehicle and omitting the damage in the application to file a claim at a later date). For this reason, insurance companies limit the availability of online policies and require an insurance appraiser verification of the vehicle's conditions. This step successfully avoid frauds, but limits

the user's engagement opportunities and the user experience. In addition, manual inspection are expensive, slow and cause of operative risks and heterogeneous evaluations.

This project aims to leverage deep learning techniques to develop a suite for automated vehicle inspection and offer market-wide coverages, currently available for already insured users only.

3.1.2 Workflow

The project is structured as it follows:

- Dataset building: the images are acquired through a scraping of damaged and not damaged car images. The scraper and the resulting dataset is reviewed in Section 2.

- Images preprocessing: scraped images are full of noise, they can not represent the desired objects or contain of several cars other than the one of interest. In order to remove this noise, the images are first passed to a neural network that identify the car and remove the background and the other present objects. Also, some data augmentation techniques are used before passing the images to the classifiers.

- CNN hyperoptimization: the first classification experiment is conducted on a convolutional neural network designed from scratch. A simple CNN, optimized with Random Search, is used as a baseline and compared to the hyperparameter set found with BOHB optimization. The experiment is detailed in Section 4.

- Transfer Learning hyperoptimization: subsequently, a transfer learning approach is followed to take advantage of pre-trained models. BOHB is used to determine the optimal cut-level of a pre-trained model, VGG16, along with

some other training hyperparameters. The resulting model is again compared to a random search model baseline. The experimental setup and the result are discussed in Section 5.

3.1.3 Hardware & Software settings

The experiments are conducted on a Amazon EC2 g4dn.xlarge instance, with a single NVIDIA T4 GPU (16 GiB memory) mounted. Although the use of a single GPU does not allow parallel evaluations of different configurations, it parallelizes the computations inside the network, giving about 30x-40x speed-up on the CNN training. The neural networks are built with keras using tensorflow backend.

3.2 Dataset

The lack of suitable datasets made it necessary to find an alternative way for obtaining the data. Web scraping was the better compromise between images quality and speed at which images are gathered.

Specifically, a binary class dataset was built, where the positive class refers to damaged cars and the negative class to not damaged cars.

The use of web scraping allowed to gather images of cars relevant to specific markets. The objective of this thesis is to provide a business application to the Italian market (at the moment), the images were scraped for the 10 most common car sold in Italy in 2020 [72].

Fiat 500X wasn't used as research keyword, for the redundancy with *Fiat 500* that would have led to redundant images, while *Dacia Sandero* was removed for the lack of images for this model's damaged cars. Other models were involved to enrich the dataset with heterogeneous type of cars.

Two scrapings were conducted for each model following these rules:

- damaged car images are obtained using the keywords *car model + "sinistrata"*

(damaged);

- not damaged cars are obtained using the keywords *car model + "seconda mano"* ("second hand"). The keyword "seconda mano" was used to avoid cover pictures of the cars, unrealistic for our business case.

The final dataset counts 5247 images, 1643 for damaged cars and 3604 for not damaged cars, as shown in Table 3.1.

Table 3.1: Dataset composition

Model	Damaged Cars	Not Damaged Cars
Fiat Panda	154	227
Lancia Ypsilon	109	362
Renault Clio	155	242
Jeep Renegade	97	241
Fiat 500	190	228
Volkswagen T-Roc	47	209
Citroen C3	95	240
Dacia Duster	133	229
Audi A3	22	186
BMW Series 3	61	156
Fiat Punto	137	241
Mercedes Class A	82	125
Opel Corsa	91	251
Volkswagen Golf	87	248
Jeep Compass	110	189
Toyota Yaris	73	230
Total	1643	3604

3.3 Images preprocessing

The images are the *fuel* of machine learning *engines*, so good quality images are needed to perform effectively. In the context of image classification, the quality of an image is an ambiguous concept. We need hetereogeneous images in terms of content (e.g. model and color of the car) to obtain generalization, but also images acquired in uniform conditions, in order to reduce the noise on the images.

Web scraping provided a good solution for heterogenous car models, at the cost of no control over the acquisition process (where *acquisition process* regards the conditions at which the photo was taken). The resulting images are very noisy and some operations are needed to clean the images.

The noise may be present in several respects. Zoom, lightness, rotation, background are only some examples of noise. Most of this aspects have been successfully addressed by using Data Augmentation, introduced with AlexNet [16], that is also useful to enlarge the training data.

3.3.1 Image Segmentation

The business nature of this application requires a clear delivery of the results. The commissioners are concerned about the ability of the model to distinguish the presence of the damage focusing only on the car, without being affected by other elements present in the images.

All the pixels outside the car itself are called *background* and needs to be carefully treated. The idea is to remove all the background and maintain only the pixels regarding the car, in order to remove all the *background noise*. This is obtained by using Mask-RCNN (described in Section 1.2.3).

No relevant reference in literature suggests that the background removal could help the classifier to improvr the model performances, especially when the training data are *big* enough. However, this task is addressed for the business needs.

The implementation of Mask R-CNN in this experiments follows the repository shared by Matterport Inc. [73]. Although this library is made on the Mask R-CNN paper for the most part, there are some differences:

- Image resizing: the images are all resized to the same size in order to support multiple images training per batch.
- Bounding Boxes: instead of using the bounding boxes provided with the datasets (when available), the bounding boxes are made on the fly. This makes it easy to apply image. augmentation and to train on multiple datasets (which some can provide bounding boxes and others not).
- Learning rate: the original paper proposed a learning rate of 0.02 which is decreased by 10 at the 120k iteration. This was found to be too higher, so a lower learning rate is used.

The library provides a pre-trained set of weights for Mask R-CNN trained on the Microsoft Common Objects in Context (COCO) dataset [74]. This model is able to identify and segment more than 80 different objects, including cars. For this reason the re-training from scratch of the network was considered unnecessary.

The computational time for car segmentation is quite high on CPU ($\sim$ 15 seconds for image), while it is almost unremarkable on GPU (less than 0.5 seconds per image).

The validation of the masking process revealed some tricky cases, including images with multiple cars. In this case, a new masked image for each detected car is created, introducing misleading cars to train on. To avoid this exception and maintain only one masked image per input image, a rule was imposed limiting the acceptation of the masked image to the greater car detected (the one of interest) with a minimum threshold for the number of pixels covered by the car (in the case the principal car would not have been detected and one background car would have been masked). An example is shown in Figure 3.1.

(a) Input image

(b) Masked cars

(c) Output image

Figure 3.1: Multiple cars detected (b) in the same input image (a). Only the biggest is maintained for the output (c).

The masking process discarded about 2,000 pictures due to a failure in the segmentation or to the rejection of the above consistency criteria. Such a high number of discarded images is due to the low-quality scraped images, which are often misleading and don't refer to the keywords used (some images do not depict any car, but advertisements or specific components). In addition to the background removal, this process has resulted in a automated cleaning of the dataset, that otherwise would have required a manual revision.

3.3.2 Dataset Split

The remaining 3018 images have been split in training (70%), validation (15%) and test (15%) datasets. The split was designed in order to keep for all the resulting datasets the damaged/not damaged ratio of the entire dataset, that is, respectively, 40% and 60%.

Although the data are not perfectly balanced, the gap between positive and negative class is not so marked. No correction is applied to the model to deal with the unbalancing, however the models are evaluated with respect to metrics that take into account the balancing in the data. precision, recall and F-measure.

The composition of the datasets is shown in Table 3.2.

Table 3.2: Train, Validation, Test datasets composition

Dataset	Damaged Cars	Not Damaged Cars
Training	846	1266
Validation	181	271
Test	182	272
Total	1206	1809

3.3.3 Data Augmentation

In order to deal with most of the image noise, augmenting the dataset volume and make the classifier more robust with respect of some secondary characteristics of the objects in the image, like zoom, rotation and shift, a Data Augmentation is applied to the train dataset. Such data augmentation is limited only to the train dataset in order to evaluate the classifier performances only on non-doctored images.

The data augmentation operations are summarized in Table 3.3.

Table 3.3: Data Augmentation

Parameter	Value
horizontal flip	True
width shift range	0.2
heigh shift range	0.2
rotation range	30
zoom range	0.15

3.4 CNN experiment

The first BOHB hyperparameters optimization experiment is conducted on a convolutional neural network build from scratch. The aim of this experiment is to evaluate the performances of BOHB hyperoptimization compared with a random search baseline. Both architecture and training hyperparameters have been used for the hyperoptimization, requiring the least possible involvement of human decisions in the process, as per the objective of AutoML. The hyperparameters under study are of mixed nature (discrete, continuous and categorical), so the use of a KDE estimator becomes necessary (see Section 2.3.4).

The number of convolutional blocks (convolutional layer + pooling layer) is the first hyperparameter to be optimized in a range [3, 5], along with the number of neurons in each layer [4, 64]. The number of neurons in the fourth and fifth convolutional layers are conditioned on the number of layers. For example, the number of neurons in the fourth conv layer are used only if the number of layers is greater than three. The first 3 layers are always present in order to learn the low-level features, that is a property for all CNNs independently of the task as stated by [31]. Then the depth of the network strongly depends on the task, so the last layers are under hyperoptimization. The activation function used in convolutional layers is ReLU (Rectified Linear Units). Since ReLU was introduced in convolutional neural networks in 2012 with Alexnet, all the following studies used such activation function for its strenghts (see Section 2.3 for more details on ReLU). Because of this wide adoption of ReLU, it was used without any hyperoptimization. The other hyperparameters of the convolutional blocks are the kernel size (3x3) of the filters and the receptive field of pooling layers (2x2).

The output of last convolutional block of the network is flattened and connected to a fully-connected component. It is made of 2 fully connected layers and a dropout layer between them. Again, the activation function of dense layers is ReLU, while the number of neurons for each layer is optimized in the range [8, 256]. The last

architecture hyperparameter used for HPO is the dropout rate optimized in [0.0 , 0.5]. The output layer is composed of one neuron and a sigmoid activation function, defined as it follows:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{3.1}$$

The sigmoid function is particularly suitable for binary classification where the output must have values in range [0,1]. Also, the sigmoid transformation ensures a strong gradient for backpropagation.

The architecture of the CNN is shown in Figure 3.2.

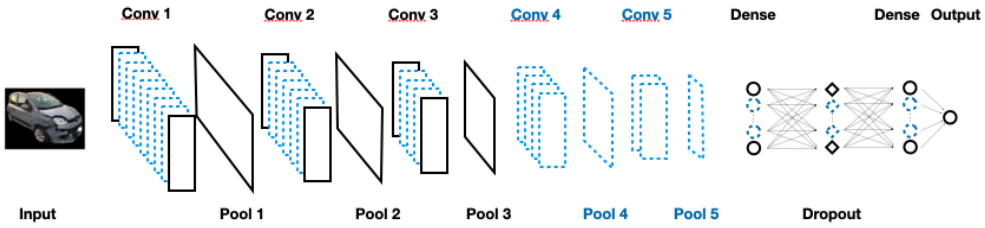

Figure 3.2: CNN architecture. The components under HPO, like the number of filters in the first conv layer and the number of Conv layers after the third layer, are indicated in blue. The components in black are not optimized (e.g. the presence of the f-c layers).

For the training side, the loss function is the binary crossentropy, while the optimizer is chosen through HPO between Adam and Stochastic Gradient Descent (SGD).

Adam (adaptive moments) is an algorithm with adaptive learning rate composed of a variation of RMSProp and momentum (directly embedded in the estimation of first-order and second-order moments). According to Kingma *et al.* [75], the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters".

The performances of Adam are controlled by the learning rate, the exponential decay rate for the first moment estimates (beta 1), the exponential decay of the

second moment estimates (beta 2) and epsilon, a constant for numerical stability. The beta hyperparamaters are left to default (that is, respectively, 0.9 and 0.999), while the learning rate and the epsilon are used in the HPO. In particular, the default value of epsilon was found to be not optimal, specially for image classification tasks (according to Keras documentation [76]). Because of the importance of this hyperparameter on the convergence of the algorithm, it was used for HPO in the range [1e-7, 1e-1] along with the learning rate [1e-3, 1e-1].

Table 3.4: Hyperparameters for CNN experiment

	Hyperparameter	Range	Type	Conditioned
h_1	# conv layers	[3, 5]	integer	no
h_2	# neurons layer 1	[4, 64]	integer	no
h_3	# neurons layer 2	[4, 64]	integer	no
h_4	# neurons layer 3	[4, 64]	integer	no
h_5	# neurons layer 4	[4, 64]	integer	$h_1 > 3$
h_6	# neurons layer 5	[4, 64]	integer	$h_1 > 4$
h_7	Dropout rate	[0, 0.9]	float	no
h_8	# f-c neurons 1	[8, 256]	integer	no
h_9	# f-c neurons 2	[8, 256]	integer	no
h_{10}	Optimizer	{Adam, SGD}	categorical	no
h_{11}	Learning rate	[1e-3, 1e-1]	float	no
h_{12}	Momentum	[0.0, 0.99]	float	$h_{10} = \text{SGD}$
h_{13}	epsilon	[1e-7, 1e-1]	float	$h_{10} = \text{Adam}$

SGD (Stochastic Gradient Descent) is one of the most famous optimizer for neural networks. It uses minibatches of data to find the direction of research. In this experiment the momentum is applied to the SGD, accelerating gradient descent in the relevant direction and dempening oscillation. The learning rate is optimized

in the same range of Adam {1e-3, 1e-1}, while the momentum is optimized in the range {0, 0.99}.

All the hyperparameters optimized, their type, range and conditioning can be found in Table 3.4.

The training augmented images are passed through minibatches of 32 examples. The network is trained for 8 epochs in the case of low-fidelity runs and 25 epochs for maximum budget runs. The budget for the whole HPO experiment is set to evaluate 24 brackets of configuration, with each bracket containing 2 or 3 configurations. The best low fidelity configuration for each bracket is then advanced to the maximum budget training (25 epochs). The results are evaluated and compared to the random search baseline, whose performances are discussed in the following subsection. The configuration performance are compared in terms of accuracy, recall, precision and F1-measure.

3.4.1 Random Search Baseline

The random search hyperparameters ran for 4 hours evaluating 25 different configurations for 25 epochs each.

The evolution of loss function over time shows good anytime performances for random search, as good configurations are found in the initial iterations (see Figure 3.3). The best (incumbent) configuration (Table 3.5) is composed of the maximum number of convolutional layers (5), meaning that a deeper network fits better with this task. It achieves 0.70137 of accuracy on validation set and 0.685022 on test set.

The recall (0.79 on validation set and 0.82 on test set) is higher than the precision (0.74 both on validation and test set) meaning that the not damaged cars wrongly classified are less than the not damaged ones.

The overall F1 score is 0.76 on validation set and 0.78 o test set.

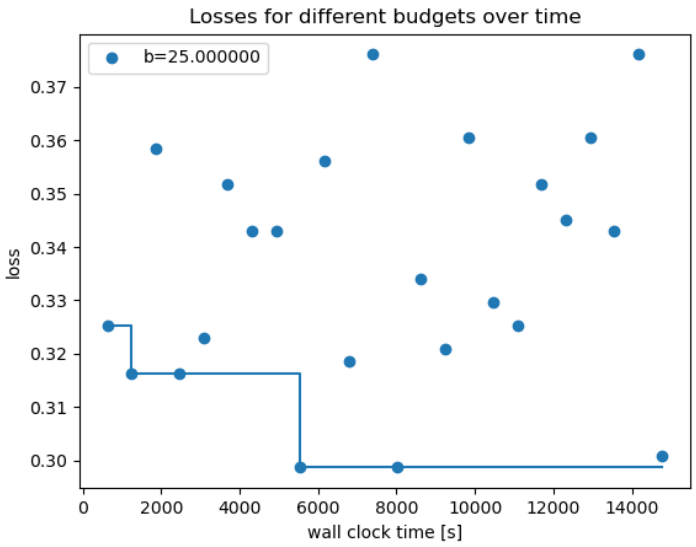

Figure 3.3: Random search HPO loss evolution on CNN from scratch.

Table 3.5: CNN incumbent configuration with random search

	Hyperparameter	incumbent
h_1	# conv layers	5
h_2	# neurons layer 1	62
h_3	# neurons layer 2	39
h_4	# neurons layer 3	10
h_5	# neurons layer 4	43
h_6	# neurons layer 5	61
h_7	Dropout rate	0.1251
h_8	# f-c neurons 1	74
h_9	# f-c neurons 2	127
h_{10}	Optimizer	SGD
h_{11}	Learning rate	9.4392e-3
h_{12}	Momentum	0.3802

3.4.2 BOHB hyperparameters optimization

For the BOHB experiment 2 more hours of computation were allocated in order to evaluate a significant number of model-based configurations (the ones suggested by the TPE). Given 12 (n) hyperparameters to be optimized, BOHB requires 13 ($n+1$) random configurations evaluations on each budget before fitting the TPE.

The use of multifidelities allows to evaluate much more configurations than random search. 57 different configurations have been evaluated for a total of 69 iterations. The discrepancy between the number of configurations and the iterations of BOHB is due to the fact that, for each bracket of low-fidelities configurations, the best one is advanced to the maximum budget.

The incumbent configuration (Table 3.6) achieved 0.712389 accuracy on validation set and 0.718062 accuracy on test set, showing slightly better final time performances than random search incumbent configuration.

Table 3.6: CNN incumbent configuration with BOHB

	Hyperparameter	incumbent
h_1	# conv layers	4
h_2	# neurons layer 1	37
h_3	# neurons layer 2	28
h_4	# neurons layer 3	7
h_5	# neurons layer 4	11
h_7	Dropout rate	0.1232
h_8	# f-c neurons 1	187
h_9	# f-c neurons 2	22
h_{10}	Optimizer	SGD
h_{11}	Learning rate	2.2008e-3
h_{12}	Momentum	0.7662

The discrepancy between recall (0.89 on validation set and 0.92 on test set) and precision (0.68 both on validation and test set) is more stark compared to the model found using random search, while the F1-score is almost the same (0.77 on validation set and 0.78 on test set).

The losses evolution in Figure 3.4 shows a greater concentration of min budget configurations in high-level loss space, while max budget configurations are concentrated in the low-level loss space, meaning that the budget values chosen are consistent with the design of multifidelities approach (low-budget configurations as performance proxies and max-budget configurations for final performances).

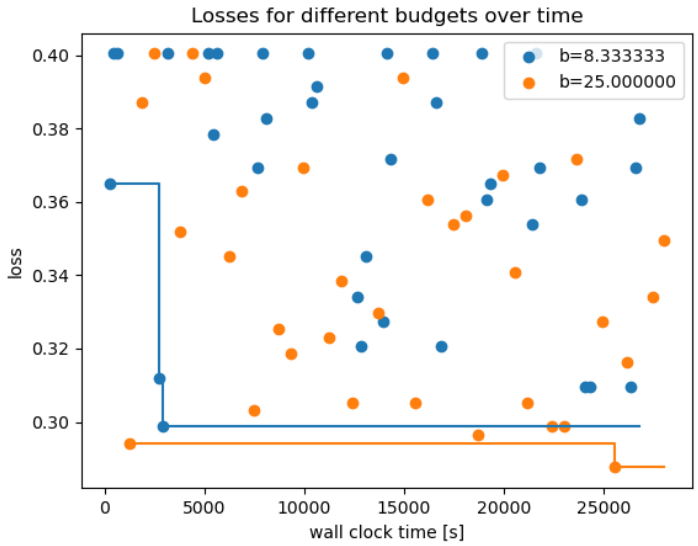

Figure 3.4: BOHB HPO loss evolution on CNN from scratch.

56 configurations have been selected at random, while 13 configurations have taken advantages of the Bayesian Process approximated by the TPE. As shown in Figure 3.5 the Bayesian Process help to exploit promising regions of the search space, as the model based configurations results in lower losses for the maximum budget.

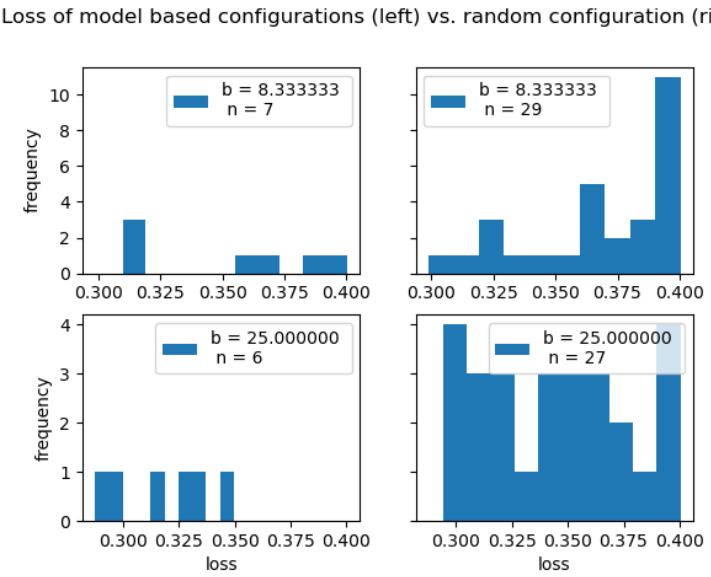

Figure 3.5: Losses comparison between model based configurations (left)and random configurations (right) on CNN from scratch

3.5 Transfer Learning experiment

Given the limited size of the dataset and the features transfearability property of CNNs discussed in Section 1.3.4, another experiment was conducted using VGG-16 as pretrained network.

The transfer learning approach restricts the architectural choices but introduces some other hyperparameters. As analysed in Section 1.3.3 the pretrained networks can be used as features extractors (freezing all the convolutional layers) or fine-tuned according to the new task (unfreezing some convolutional layers). The *cut level*, meant as the last layer kept freezed, is the hyperparameter that juggles between feature extraction and fine-tuning.

The convolutional layers of VGG16 are extracted and used as pretrained network, to which it is connected a fully connected and a dropout layer. Both the neurons of the first fully connected layer and the dropout rate are optimized with BOHB. Another fully-connected layer with dropout is connected when the number of dense

layers is greater than one.

The training of the unfrozen layers requires some more caution, especially when some convolutional layers are unfrozen and we want to fine-tune them. In this case, the fully connected layers placed on top of the network need to adapt themselves to the already learned weights of the convolutional layers before unfreezing these conv layers. This is obtained by pre-training only the head of the network (the fully-connected component) with a very small learning rate (1e-4) and then unfreezing the convolutional layers, eventually increase the learning rate and continue the training. This procedure, often referred as *warm-starting* of the network, avoid the disruption of the learned weights through backpropagation, but rather slowly adapt the weights to the new task [28]. The pretraining phase lasts 10 epochs, then the training follow with other 5 epochs for minimum budget allocations and 15 epochs for maximum budget allocations.

Table 3.7: Hyperparameters for TL experiment

	Hyperparameter	Range	Type	Conditioned
h_1)	# f-c layers	[1, 2]	integer	no
h_2)	Dropout rate 1	[0, 0.5]	float	no
h_3)	Dropout rate 2	[0, 0.5]	float	$h_1 > 1$
h_4)	# f-c neurons 1	[8, 256]	integer	no
h_5)	# f-c neurons 2	[8, 256]	integer	$h_1 > 1$
h_6)	Optimizer	{Adam, SGD}	categorical	no
h_7)	Learning rate	[1e-3, 1e-1]	float	no
h_8)	Momentum	[0.0, 0.99]	float	$h_6 = \text{SGD}$
h_9)	epsilon	[1e-7, 1e-1]	float	$h_6 = \text{Adam}$
h_{10})	cut-level	{7, 11, 15, 20}	categorical	no

VGG16 is made of 5 convolutional blocks (as illlustrated in Section VGG16).

The first two convolutional blocks (each constituted of 2 convolutional layers and one max pooling layer) are always kept frozen, as they are responsible of the most general features learning. The cut level can occur at the end of each subsequent convolutional block: at the 7th, the 11th, the 15th and the 20th layer. If the cut level is chosen at the 20th layer, all the convolutional layers remain frozen, so VGG16 act as feature extractor. In the other cases the pre-trained network is fine-tuned.

The hyperparameters under optimization are summarized in table 3.7.

3.5.1 Random Search Baseline

Random search ran for 6 hours evaluating 31 configurations and it achieved 0.853982 accuracy on validation set and 0.845815 accuracy on test set. As for the CNN from scratch optimization, the recall is definitely higher than precision both on validation set ($p = 0.83$, $r = 0.96$) and test set ($p = 0.82$, $r = 0.93$). The F1-score is equal to 0.89 on validation set and 0.87 on test set.

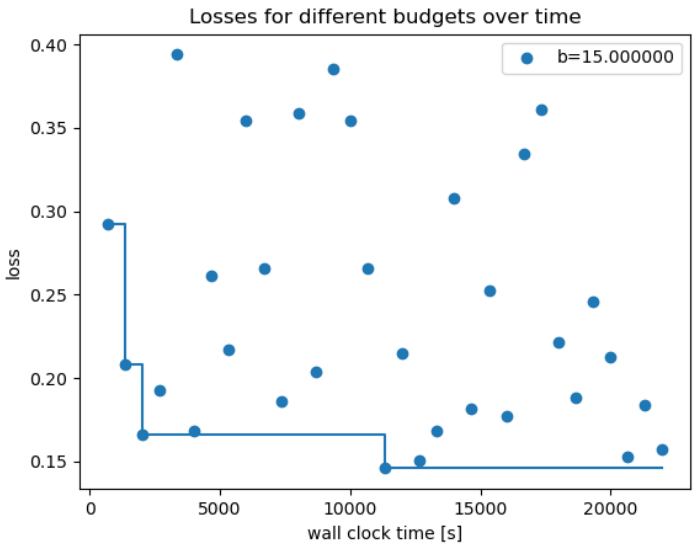

Figure 3.6: Random search HPO loss evolution on Transfer learning.

Table 3.8: TL incumbent configuration with random search

	Hyperparameter	incumbent
h_1	# f-c layers	2
h_2	Dropout rate 1	0.1875
h_3	Dropout rate 2	0.1851
h_4	# f-c neurons 1	132
h_5	# f-c neurons 2	27
h_6	Optimizer	Adam
h_7	Learning rate	8.14653e-4
h_9	epsilon	7.56647e-3
h_{10}	cut-level	15

The best configuration (Table 3.8) sets the cut level to the 15th layer, unfreezing only the last convolutional layer. As for the CNN from scratch experiment, random search shows good anytime performances (Figure 3.6).

3.5.2 BOHB hyperparameters optimization

BOHB hyperoprtimization for transfer learning is the core experiment of this thesis, so the maximum amount of resources in terms of training time were allocated in this step. In 11 hours of computations 63 different configurations have been evaluated on a total of 76 runs (29 of which related to model-based picks).

The anytime performances are quite the same as random search, while the final performances are better, with 0.867257 accuracy on validation set and 0.88784 accuracy on test set. The errors results balanced, with very close values between precision and recall both on validation set ($p, r = 0.89$) and test set ($p = 0.89, r = 0.91$). Also, the F1-score is the highest reached in all experiments, 0.89 on validation set and 0.90 on test set.

Although the computational time allocated to BOHB doubles the one allocated to random search, the comparison is fair because random search does not take advantage of its past and 31 tries are sufficient as baseline.

As shown in Figure 3.7 there is an higher concentration of configurations with good performances than the ones found by random search, especially when the maximum budget is allocated (that is the same of random search experiment). The evolution of the losses shows a continuous improvement over time, gained by the points added to the gaussian process estimation.

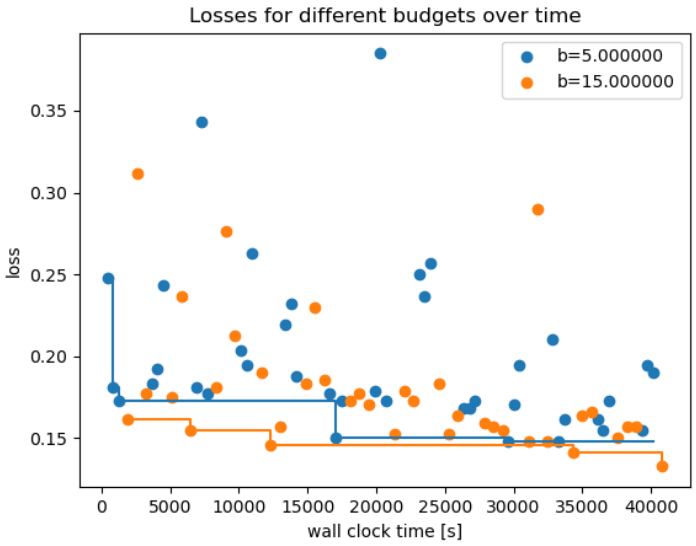

Figure 3.7: BOHB HPO loss evolution on transfer learning.

The random configurations picked by successive halving obtained almost the same performance than random search, but with less low-performing configurations. Once enough configurations on each budget have been evaluated, the new proposals are made by the Gaussian process approximated by the TPE.

The Gaussian process fits well the loss function associated to the model, finding better solutions than random picks. Also, it found the best configuration improving by 4% of accuracy on test set the best solution found by successive halving and random search.

The incumbent configuration (Table 3.9) consists of only one dense layer before the output layer (along with a dropout layer) and one more unfrozen convolutional block than the random search incumbent configuration.

Table 3.9: TL incumbent configuration with BOHB

	Hyperparameter	incumbent
h_1	# f-c layers	1
h_2	Dropout rate 1	0.0196
h_4	# f-c neurons 1	208
h_6	Optimizer	Adam
h_7	Learning rate	4.65803e-4
h_9	epsilon	3.17692e-4
h_{10}	cut-level	11

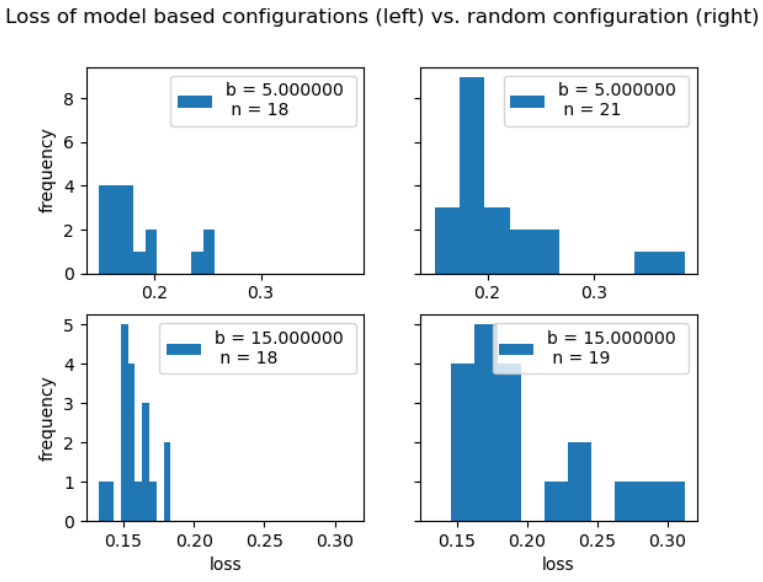

Figure 3.8: Losses comparison between model based configurations (left) and random configurations (right) on transfer learning

3.6 Results comparison and discussions

The results of each hyperoptimization experiment are summarized in Table 3.10 for validation set and Table 3.11 for test set.

Table 3.10: Results comparison on validation set

Experiment	Accuracy	Precision	Recall	F1-measure
Random Search on CNN	0.701	0.740	0.790	0.760
BOHB on CNN	0.718	0.680	0.890	0.770
Random Search on TL	0.854	0.830	0.960	0.890
BOHB on TL	0.867	0.890	0.890	0.890

Table 3.11: Results comparison on test set

Experiment	Accuracy	Precision	Recall	F1-measure
Random Search on CNN	0.685	0.740	0.820	0.780
BOHB on CNN	0.719	0.680	0.920	0.780
Random Search on TL	0.846	0.820	0.930	0.870
BOHB on TL	0.888	0.890	0.910	0.900

The use of the pre-trained model results in better anytime and final performances compared to the CNN from scratch, independently on the hyperparameter optimization algorithm applied. The small size of the training data limit the learning power of the network, that does not exceed the 70% of accuracy.

On the contrary, the already learned filters of VGG16 helps to use finer features, improving all the classification metrics.

BOHB shows a slightly better final time performances than random search optimizing the CNN from scratch, but not as much as desired. The object function

seems to be quite flat around 70% of accuracy, since no substantial improvements are gained by changing the architecture nor the learning hyperparameters. Actually, as shown in Figure 3.4, the accuracies obtained by the 57 different configurations ranges in only 10 percentage points.

The advantages of Bayesian optimization next to Hyperband is more noticeable for the transfer learning experiment, where the incumbent configuration found through BOHB has a 5% better accuracy than the model configured through random search. In this case the random search appears to be more complex and different hyperparameters settings lead to very different performances. Both the optimal configurations found by random search and BOHB fine-tune the pre-trained network, unfreezing respectively the latest and the last two convolutional blocks.

It is not surprising that the cut level is somewhere between the earliest convolutional layers and the end of the convolutional part of the model: the poor performances of the CNN from scratch suggested how training most of the network would have not led to satisfying results, while the difference between the original task on which VGG16 was trained (objects classification on Imagenet) and the target task (damage detection) makes the feature extraction not the best solution.

Conclusions

The application of convolutional neural networks on damage detection has revealed how much the volume of training data is important to leverage all their potential. The CNN from scratch was not able to extract useful features from the images, while the use of a pretrained network has compensated the lack of training images.

Optimizing the hyperparameters of a CNN with limited power, as the CNN from scratch, did not provide any substantial improvement to the classifier performances, neither using random methods or BOHB.

On the contrary, hyperoptimization results on transfer learning showed the potential of BOHB when applied to performing workflows. BOHB fulfilled all the HPO techniques desiderata, making it an *efficient* and *effective* solution.

Further improvements in terms of efficiency can be obtained by modifying the brackets allocation during the iterations, favouring larger brackets in the earliest iterations (the ones needed to initiate the Gaussian process) and smaller brackets when the model-based picks are available.

Before delivering this solution to the market, some refinements are needed to improve the robustness of the algorithm with respect to the damage type, car model and image acquisition process. Furthermore, the training images used in these experiments refer to severe damages, which makes the algorithm not suitable for milder damages, like scratches. Also, labelling images for the type of damage, could give the robustness needed and extend the model capabilities to the damage tagging.

Bibliography

[1] Alexander Andreopoulos and John K. Tsotsos. "50 Years of object recognition: Directions forward". In: *Comput. Vis. Image Underst.* 117 (2013), pp. 827–891.

[2] R.Graves. *The Greek Myths: Complete Edition, Penguin, 1993.*

[3] David H. Hubel and Torsten N. Wiesel. "Receptive Fields of Single Neurons in the Cat's Striate Cortex". In: *Journal of Physiology* 148 (1959), pp. 574–591. URL: https://www.bibsonomy.org/bibtex/202c5cf1ee910eadba5efa77b3cd043f6/idsia.

[4] Madhusmita Sahu and Rasmita Dash. "A Survey on Deep Learning: Convolution Neural Network (CNN)". In: Jan. 2021, pp. 317–325. ISBN: 978-981-15-6201-3. DOI: 10.1007/978-981-15-6202-0_32.

[5] *Hubel & Wiesel experiment.* [Online; accessed 30-December-2020]. URL: https://goodpsychology.wordpress.com/2013/03/13/235/.

[6] Wikipedia contributors. *Russell Kirsch — Wikipedia, The Free Encyclopedia.* [Online; accessed 31-December-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Russell_Kirsch&oldid=973385657.

[7] Rostyslav Demush. *A Brief History of Computer Vision (and Convolutional Neural Networks).* [Online; accessed 30-December-2020]. URL: https://hackernoon.com/a-brief-history-of-computer-vision-and-convolutional-neural-networks-8fe8aacc79f3.

[8] K. Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36 (2004), pp. 193–202.

[9] Kunihiko Fukushima. "Recent advances in the deep CNN neocognitron". In: *Nonlinear Theory and Its Applications, IEICE* 10.4 (2019), pp. 304–321. DOI: 10.1587/nolta.10.304.

[10] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Comput.* 1.4 (Dec. 1989), 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. URL: https://doi.org/10.1162/neco.1989.1.4.541.

[11] Yann LeCun and Corinna Cortes. *MNIST handwritten digit database*. 2010. URL: http://yann.lecun.com/exdb/mnist/.

[12] Rajesh P.N. Rao and Dana H. Ballard. "An active vision architecture based on iconic representations". In: *Artificial Intelligence* 78.1 (1995). Special Volume on Computer Vision, pp. 461 –505. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(95)00026-7. URL: http://www.sciencedirect.com/science/article/pii/0004370295000267.

[13] D. G. Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410.

[14] *The PASCAL VOC project*. [Online; accessed 31-December-2020]. URL: http://host.robots.ox.ac.uk/pascal/VOC/.

[15] J. Deng et al. *ImageNet: A Large-Scale Hierarchical Image Database*. http://www.image-net.org/papers/imagenet_cvpr09. 2009.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th*

International Conference on Neural Information Processing Systems - Volume 1. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, 1097–1105.

[17] Wikipedia contributors. *ImageNet — Wikipedia, The Free Encyclopedia*. [Online; accessed 31-December-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=ImageNet&oldid=991648926.

[18] Afshine Amidi, Shervine Amidi. *The evolution of image classification explained*. [Online; accessed 31-December-2020]. URL: https://stanford.edu/~shervine/blog/evolution-image-classification-explained.

[19] Jerry Wei. *VGG Neural Networks: The Next Step After AlexNet*. URL: https://towardsdatascience.com/vgg-neural-networks-the-next-step-after-alexnet-3f91fa9ffe2c.

[20] C. Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[21] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[22] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[23] Prakhar Ganesh. *From LeNet to EfficientNet: The evolution of CNNs*. [Online; accessed 04-January-2021]. URL: https://towardsdatascience.com/from-lenet-to-efficientnet-the-evolution-of-cnns-3a57eb34672f.

[24] A. Waibel et al. "Speech recognition using time-delay neural networks". In: *The Journal of the Acoustical Society of America* 83.S1 (1988), S45–S46. DOI: 10.1121/1.2025362. URL: https://doi.org/10.1121/1.2025362.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[26] Wikipedia contributors. *Convolution — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Convolution&oldid=1007580401. [Online; accessed 19-February-2021]. 2021.

[27] Fei-Fei Li,Ranjay Krishna, Danfei Xu, Amelie Byun. *CS231n Convolutional Neural Networks for Visual Recognition*. [Online; accessed 05-January-2021]. 2019. URL: https://cs231n.github.io/convolutional-networks/?source=post_page-----a998dbc1e79a----------------------#case.

[28] A. Rosebrock. *Deep Learning for Computer Vision with Python: Starter Bundle*. PyImageSearch, 2017. URL: https://books.google.it/books?id=9Ul-tgEACAAJ.

[29] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2015. arXiv: 1412.6806 [cs.LG].

[30] Wikipedia contributors. *Artificial neural network — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=1007557970. [Online; accessed 19-February-2021]. 2021.

[31] Jason Yosinski et al. *How transferable are features in deep neural networks?* 2014. arXiv: 1411.1792 [cs.LG].

[32] Dipanjan (DJ) Sarkar. *A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning*.

[33] Waleed Abdulla. *Splash of Color: Instance Segmentation with Mask R-CNN and TensorFlow*. [Online; accessed 13-January-2021]. URL: https://engineering.matterport.com/splash-of-color-instance-segmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46.

[34] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv: 1311.2524 [cs.CV].

[35] Rohith Gandhi. *SR-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. [Online; accessed 13-January-2021]. URL: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

[36] Ross Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV].

[37] S. Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6 (2017), pp. 1137–1149. DOI: 10.1109/TPAMI.2016.2577031.

[38] Kaiming He et al. *Mask R-CNN*. 2018. arXiv: 1703.06870 [cs.CV].

[39] *NIPS workshops*. [Online; accessed 07-January-2021]. URL: http://www.cs.cmu.edu/afs/cs/project/cnbc/nips/NIPS95/Workshops.html.

[40] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning". In: 22.10 (Oct. 2010), 1345–1359. ISSN: 1041-4347. DOI: 10.1109/TKDE.2009.191. URL: https://doi.org/10.1109/TKDE.2009.191.

[41] Qiang Yang et al. *Transfer Learning*. Cambridge University Press, 2020. DOI: 10.1017/9781139061773.

[42] Dipanjan Sarkar, Raghav Bali, and Tamoghna Ghosh. *Hands-On Transfer Learning with Python: Implement Advanced Deep Learning and Neural Network Models Using TensorFlow and Keras*. Packt Publishing, 2018. ISBN: 1788831306.

[43] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[44] Wikipedia contributors. *Universal approximation theorem — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-December-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Universal_approximation_theorem&oldid=993903484.

[45] Wikipedia contributors. *Moore's law — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=1006584730. [Online; accessed 6-March-2021]. 2021.

[46] D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems for Optimization". In: *Trans. Evol. Comp* 1.1 (Apr. 1997), 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893. URL: https://doi.org/10.1109/4235.585893.

[47] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, eds. *Automated Machine Learning: Methods, Systems, Challenges*. In press, available at http://automl.org/book. Springer, 2018.

[48] Wikipedia contributors. *Global optimization — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-December-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Global_optimization&oldid=988314442.

[49] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *J. Mach. Learn. Res.* 13.null (Feb. 2012), 281–305. ISSN: 1532-4435.

[50] Eric Brochu, Matthew W. Hoffman, and Nando de Freitas. "Portfolio Allocation for Bayesian Optimization". In: *CoRR* abs/1009.5419 (2010). URL: http://arxiv.org/abs/1009.5419.

[51] Matthias Feurer, Benjamin Letham, and Eytan Bakshy. *Scalable Meta-Learning for Bayesian Optimization*. 2018. arXiv: 1802.02219 [stat.ML].

[52] Xueli Xiao et al. *Efficient Hyperparameter Optimization in Deep Learning Using a Variable Length Genetic Algorithm*. 2020. arXiv: 2006.12703 [cs.NE].

[53] Matteo Fischetti and Matteo Stringher. *Embedded hyper-parameter tuning by Simulated Annealing*. 2019. arXiv: 1906.01504 [cs.LG].

[54] Baosu Guo et al. "The Tabu_Genetic Algorithm: A Novel Method for Hyper-Parameter Optimization of Learning Algorithms". In: *Electronics* 8.5 (2019), p. 579. ISSN: 2079-9292. DOI: 10.3390/electronics8050579. URL: http://dx.doi.org/10.3390/electronics8050579.

[55] Bowen Baker et al. *Designing Neural Network Architectures using Reinforcement Learning*. 2017. arXiv: 1611.02167 [cs.LG].

[56] Chuan Li. *OpenAI's GPT-3 Language Model: A Technical Overview*. [Online; accessed 18-December-2020]. 2020. URL: https://lambdalabs.com/blog/demystifying-gpt-3/#1.

[57] K. Jamieson and Ameet Talwalkar. "Non-stochastic Best Arm Identification and Hyperparameter Optimization". In: *AISTATS*. 2016.

[58] Stefan Falkner, Aaron Klein, and Frank Hutter. "BOHB: Robust and Efficient Hyperparameter Optimization at Scale". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, 2018, pp. 1437–1446. URL: http://proceedings.mlr.press/v80/falkner18a.html.

[59] Wikipedia contributors. *Normal distribution — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-December-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Normal_distribution&oldid=994760864.

[60] Wikipedia contributors. *Multivariate normal distribution — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-December-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Multivariate_normal_distribution&oldid=991297516.

[61] Jochen Görtler, Rebecca Kehlbeck, and Oliver Deussen. "A Visual Exploration of Gaussian Processes". In: *Distill* (2019). DOI: 10.23915/distill.00017. URL: https://distill.pub/2019/visual-exploration-gaussian-processes.

[62] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. ISBN: 026218253X.

[63] *Pythonhosted - infpy*. https://pythonhosted.org/infpy/gps.html. [Online; accessed 6-March-2021].

[64] H. Wang et al. "A new acquisition function for Bayesian optimization based on the moment-generating function". In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2017, pp. 507–512. DOI: 10.1109/SMC.2017.8122656.

[65] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. arXiv: 1807.02811 [stat.ML].

[66] H. J. Kushner. "A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise". In: *Journal of Basic Engineering* 86.1 (1964), pp. 97–106. DOI: 10.1115/1.3653121. URL: https://doi.org/10.1115\%2F1.3653121.

[67] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. "The Application of Bayesian Methods for Seeking the Extremum". In: *Towards Global Optimization* 2.117-129 (1978), p. 2.

[68] James Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS'11. Granada, Spain: Curran Associates Inc., 2011, 2546–2554. ISBN: 9781618395993.

[69] Lisha Li et al. *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*. 2018. arXiv: 1603.06560 [cs.LG].

[70] AutoML contributors. *AutoML*. [Online; accessed 22-December-2020]. URL: https://www.automl.org/blog_bohb/.

[71] Stefan Falkner, Aaron Klein, and Frank Hutter. "Combining hyperband and bayesian optimization". In: *NIPS 2017 Bayesian Optimization Workshop (Dec 2017)*. 2017.

[72] Paolo Odinzov. *Repubblica, Tutti in fila per loro: ecco le 10 auto più vendute in Italia*. [Online; accessed 11-January-2021]. URL: https://www.repubblica.it/motori/sezioni/attualita/2020/09/07/news/tutti_in_fila_per_loro_ecco_le_10_auto_piu_vendute_in_italia-266476920/.

[73] Waleed Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. https://github.com/matterport/Mask_RCNN. 2017.

[74] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].

[75] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[76] Keras contributors. *Keras API reference: Adam*. https://keras.io/api/optimizers/adam/.