DEGREE PROJECT IN ,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Hyperparameter optimization for artificial neural networks

CLÉMENT GOUSSEAU

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Hyperparameter Optimization for Artificial Neural Networks

CLÉMENT GOUSSEAU

Master in Computer Science
Date: August 2, 2019
Supervisor: Erik Fransén
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Host company: Orange Labs Lannion
Swedish title: Hyperparameteroptimering av Artificiell Neuralt Nätverk

Abstract

Training algorithms for artificial neural networks depend on parameters which are called the hyperparameters. They can have a strong influence on the trained model but they are often chosen manually with trial and error experiments.

This thesis, conducted at *Orange Labs Lannion*, presents and evaluates three algorithms that aims at solving this task: a naive approach (random search), a bayesian approach (Tree Parzen Estimator) and an evolutionary approach (Particle Swarm Optimization). A well known dataset for handwritten digit recognition (MNIST) is used to compare these algorithms.

These algorithms are also evaluated on audio classification, which is one of the main activities in the company team where the thesis was conducted.

The evolutionary algorithm (PSO) showed better results than the two other methods.

Sammanfattning

Hyperparameteroptimering är en viktig men svår uppgift vid utbildning av ett artificiell neuralt nätverk.

Denna avhandling, utförd vid *Orange Labs Lannion*, presenterar och utvärderar tre algoritmer som syftar till att lösa denna uppgift: en naiv strategi (slumpmässig sökning), en bayesisk metod (TPE) och en evolutionär strategi (PSO). MNIST-datasätt används för att jämföra dessa algoritmer.

Dessa algoritmer utvärderas också med hjälp av ljudklassificering, som är kärnverksamheten i företaget där avhandlingen genomfördes.

Evolutionsalgoritmen (PSO) visade bättre resultat än de två andra metoderna.

Acknowledgements

I would like to thank:

Lionel-Delphin Poulat, Orange Labs Lannion, for his advice, support, and kindness.

Christian Grégoire, Orange Labs Lannion, for his welcome and his trust.

Pawel Herman, KTH, for his clear and useful guidelines.

Erik Fransén, KTH, for his support.

Contents

Chapter 1

Introduction

This chapter introduces the context of this thesis, its motivation and its objective.

1.1 Background

Artificial neural networks (ANN) have become very popular in the field of machine learning, especially for tasks such as image classification. To solve its assigned task an ANN needs to be trained. For example in image classification with supervised learning, many images with their corresponding labels are given. Then a learning algorithm is used. Usually it is an optimization algorithm that iteratively modifies the parameters of the network in order to minimize some loss function.

Convolutional Neural Networks (CNN) belong to the family of Artificial Neural Networks. These are composed of convolutional filters which are stacked. These convolutional layers are very good at feature extraction which is useful for many machine learning tasks (e.g., classification, detection). CNN showed best results in recent machine learning challenges such as ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Therefore this thesis will focus on this kind of ANN.

However the performance of an ANN depends a lot on another class of parameters which are called the hyperparameters. These hyperparameters must be initialized before the training and they are not modified during the

training. They can define the topology of the network (depth, number of units by layer, size of the convolutional filters, dropout rate) or the learning configuration (learning rate, number of epochs, size of the mini-batch).

1.2 Research Question

There does not exist a general rule for the choice of the hyperparameters so they are often tuned manually based on the data scientist's experience or on some experiments. Without prior knowledge, it is difficult to know which areas of the search space will be the most promising. Therefore hand-tuning may lead to hyperparameters evaluations of limited usefulness and as a consequence, a very time-consuming process.

This thesis intends to provide people with a scientific background (students, engineers, researchers) solutions to automatically optimize hyperparameters and metrics to quantify their effectiveness in term of search-space exploration.

Chapter 2

Relevant Theory

Chapter 2 presents concepts that are essential in order to understand this thesis. It starts with basic knowledge about artificial neural networks. Then the problem of hyperparameter optimization is defined and three hyperparameter optimization algorithms from the literature are presented.

2.1 Artificial Neural Networks (ANN)

2.1.1 Artificial Neural Networks Layers

In this section, the main neural layers that are used in the experiments are reviewed.

Fully Connected Layer

A fully connected layer composed of m neurons takes as input a vector

$$x = \begin{bmatrix} x_1 \\ \cdots \\ x_{n-1} \\ 1 \end{bmatrix} \in \mathbb{R}^n$$

It returns an output vector

$$y = \begin{bmatrix} y_1 \\ ... \\ y_m \end{bmatrix} \in \mathbb{R}^m$$

such that

$$y = Wx \text{ with } W = \begin{bmatrix} w_{1,1} & ... & w_{1,n} \\ ... & ... & ... \\ w_{m,1} & ... & w_{m,n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Each of the m rows of W corresponds to a neuron and the coefficient $w_{i,j}$ defines the connection between the input x_j and the output y_i.

The term "1" in the vector x is a way to include a bias.

2D Convolutional Layer

A convolutional layer composed of n convolution filters takes as input a tensor $x \in \mathbb{R}^{w \times h \times c}$. w is the width of the tensor, h its height and c its number of channels.

For $1 \leq i \leq n$, the ith convolutional filter is a tensor $f_i \in \mathbb{R}^{w' \times h' \times c}$ (also refered as 'kernel'). The number of channels of the convolutional filters must be the same as the number of channels of the input tensor.

It returns an output tensor $y \in \mathbb{R}^{w \times h \times n}$ [1, equation 9.7].

Each convolution filter "slides" over the input. For each area of the input it returns the dot product between this area and the convolution filter. The figure below presents an example with a one-channel input. The dimensions of the convolutional filters w' and h' are usually much lower than the dimensions of the input w and h. Therefore a convolutional layer contain much fewer parameters than a fully-connected layer. The smallness of the filters also enables to extract local properties of the input (lines, corners, blobs,etc.). This is useful for feature extraction.

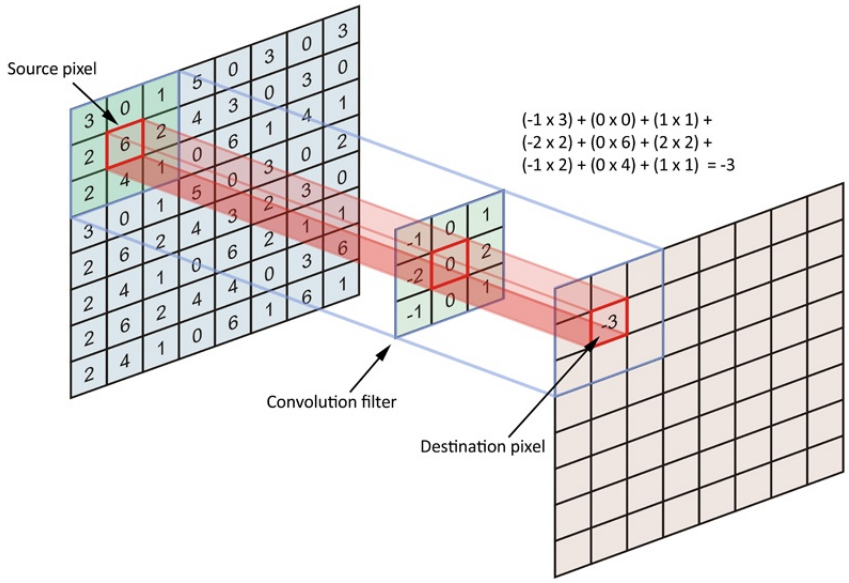


Figure 2.1: Example of a convolution with one channel [2].

Pooling Layer

Usually for the task of ANN (image classification, sound tagging, etc.) the input space is a space of high dimension. For instance, a 32×32 pixels image is a vector of $\mathbb{R}^{32\times32}$ which is equivalent to $\mathbb{R}^{1024}$ (relation of bijection). On the other hand, the output space is usually a space of low dimension (number of possible classes, number of possible tags, etc.).

Convolution layers increase the dimension of the input if the number of convolution filters is greater than the number of channels. Therefore it is useful to add layers which reduce the dimension of the data. Pooling layers are meant for that.

Like convolution layers, a pooling layer "slides" over the input. For each area of the input it returns a single value, and therefore reduces the dimension of the input. This value can be the mean of the area (average pooling), the maximum of the area (max pooling), etc.

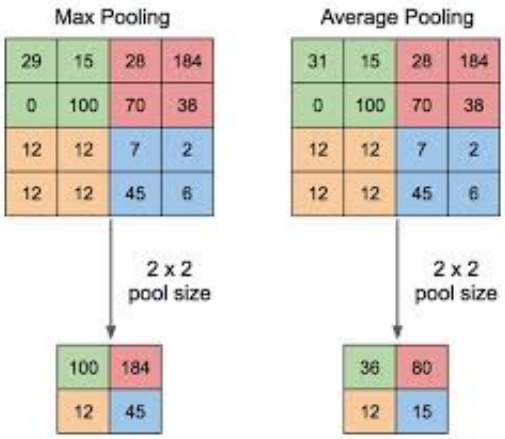


Figure 2.2: Max Pooling and Average Pooling [3].

Activation Layer

Fully-connected layers and convolution layers are linear transformations. Stacking fully-connected layers would result in another fully-connected layer, which would also be a linear transformation. The purpose of activation is to introduce non-linearity in ANN in order to approximate non-linear functions. Different activation functions exist:

- $sigmoid : x \in R \longmapsto \frac{1}{1+e^{-x}} \in [0, 1]$
- $tanh : x \in R \longmapsto \frac{1-e^{-2x}}{1+e^{-2x}} \in [-1, 1]$
- $relu : x \in R \longmapsto \max(0, x) \in [0, +\infty[$

A common problem with $sigmoid$ and $tanh$ is that they saturate for large input values which may cause problems for the gradient backpropagation (see 2.1.2). $relu$ is meant for avoiding that.

Batch Normalization Layer [4]

During the training of an ANN (see 2.1.2), the parameters of the layers are modified. As a consequence, the distribution of the input of each layer changes during the training, this is the "internal covariance shift". Batch Normalization normalizes the input data of each layer to solve this problem.

Batch Normalization also helps the input values of activation functions to avoid saturation areas of these functions.

Flatten Layer

This is a simple layer that takes a tensor as input and flattens it to output a vector. This layer is usually placed before a fully-connected layer since fully-connected layers take vectors as input.

2.1.2 Artificial Neural Networks Training

Supervised Learning

In this thesis, all the machine learning tasks are done in a context of supervised learning. This means the used data contain both input data (images, sounds, etc.) and their corresponding labels (class, tags, etc.).

Loss Function

The purpose of the loss function is to quantify the error between the true output y_{true} and and the prediction of the ANN y_{pred}.

In the the case of a multi-label classification task, y_{true} and $y_{pred} \in \mathbb{R}^{n \times d}$ where n is the number of samples and d the number of labels. $y_{true_{i,j}}$ is 1 if the label j is present in the sample i, 0 if not. $y_{pred_{i,j}}$ is the probability of presence of the label i in the sample j.

There are different loss functions. Two widely used loss functions are:

Binary Crossentropy

$bce(y_{pred}, y_{true}) = -\frac{1}{nd} \sum_{i=1}^{n} \sum_{j=1}^{d} [y_{true_{i,j}} \log(y_{pred_{i,j}}) + (1 - y_{true_{i,j}}) \log(1 - y_{pred_{i,j}})]$

This loss function will be optimized during the training (see 2.1.2).

Optimization of the Loss Function: the Stochastic Gradient Descent (for supervised learning)

The goal of the training is to minimize the loss function L_W on a training set with respect to the parameters of the ANN W. First the parameters W are initialized randomly. Then iteratively, inputs x_i and their corresponding labels y_i are randomly sampled and $\nabla_W L_W(x_i, y_i)$ the gradient of the loss in (x_i, y_i) with respect to W is computed. The weights W are updated in the direction of this gradient in order to minimize the loss.

Inputs: $X = x_1, ..., x_n$, the input data
$Y = y_1, ..., y_n$, the labels
η, the learning rate,
$epochs$, the number of iterations
Result: W, the model parameters
Initialize randomly W the parameters of the ANN
for $i \leftarrow 1$ **to** $epochs$ **do**
  Sample (x_i, y_i) from (X, Y)
  Update the parameters: $W = W - \eta \nabla_W L_W(x_i, y_i)$
end
return W

Figure 2.3: Stochastic Gradient Descent Algorithm

η is the learning rate, which defines "how much" is learned at every update. A low learning rate may result in a slow learning whereas a high learning rate may result in an unstable learning.

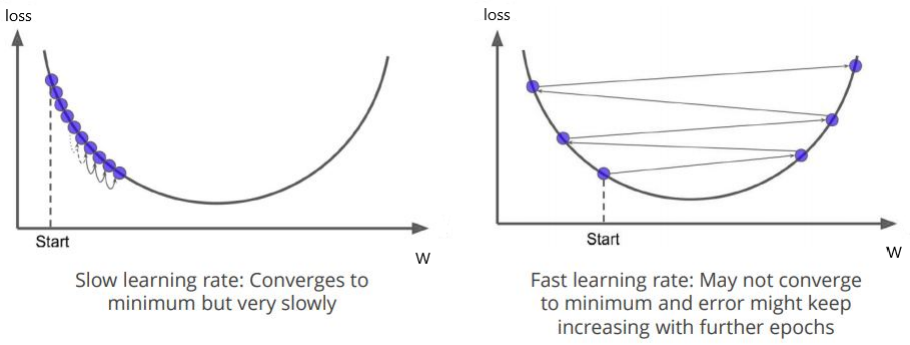


Figure 2.4: Impact of the learning rate on the training [5].

Stochastic gradient descent can also be improved by introducing a momentum: at each update the parameters W are updated by a moving average of the current gradient and the past gradients. This enables to converge quickly and with less oscillations towards a local minimum of the loss function. *Adam* [6] is one algorithm that includes this notion of momentum.

Regularization Techniques

A common problem in training an ANN is when the network performs very well on the training data but poorly on unseen data. This is overfitting. There are several methods to avoid overfitting and build more robust networks.

Batch Learning
Instead of sampling one by one input data during the stochastic gradient descent, data are sampled by batch. Then at each the gradient of the loss is computed on a whole batch. This enables a more stable learning since it averages the gradient on several inputs and may attenuate the impact of very specific data.

Weight Decay
During the training, some weights of the ANN may become very large and this may be a sign of overfitting. Weight decay consists in adding a penalty term to the loss function so that the weights do not become to large.

In the case of an ANN with one fully-connected layer, with an input x, a predicted output $y_{pred} = Wx$ (see 2.1.1), and a true output y_{true}, and *loss* as loss function, the loss becomes

$$loss(y_{pred}, y_{true}) + \beta \|W\|_2^2$$

with

$$\|W\|_2^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} w_{i,j}^2 \text{ given } W = \begin{bmatrix} w_{1,1} & \dots & w_{1,n} \\ \dots & \dots & \dots \\ w_{m,1} & \dots & w_{m,n} \end{bmatrix} \text{ (l2-norm)}$$

Other norms (e.g., l1-norm can used).

β is a hyperparameter that quantifies the power of the regularization: a large β means a strong regularization.

Dropout [7]
During the training, some connections between neurons are temporarily set to zero. This enables to build a more robust network since it forces the network to let the information flow through different paths. The *dropout rate* defines the proportion of connections which will be set to zero at each update.

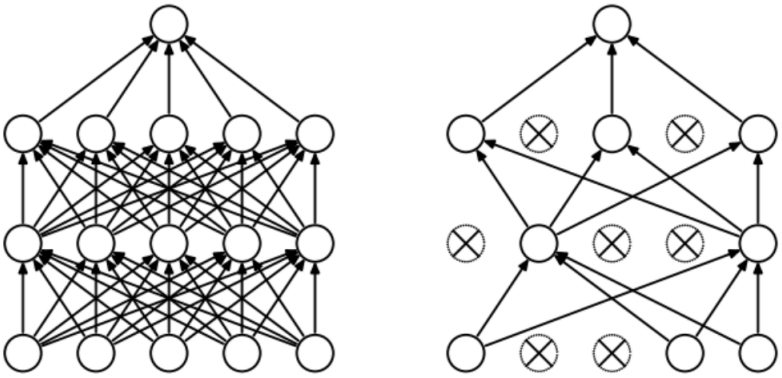


Figure 2.5: an ANN with and without dropout [7].

2.2 The Problem of Hyperparameter Optimization

2.2.1 Definition of the problem

In a typical classification problem with ANN, the data is split into a training set X_{train} and a validation set $X_{validation}$.

The goal is to find the parameters W which minimizes a loss function $L(W|X_{validation})$ on the validation set $X_{validation}$. The parameters W are obtained by a learning algorithm A defined by its hyperparameters λ and trained on the training set X_{train}. Then $W = A(X_{train}|\lambda)$. The goal of hyperparameter selection is to find λ which minimizes $L(W|X_{validation})$

[8]:

$$\lambda^* = \arg\min_{\lambda} L(W|X_{validation}) = \arg\min_{\lambda} L(A(X_{train}|\lambda)|X_{validation}) \quad (2.1)$$

It is important to understand the difference between the parameters W and the hyperparameters λ. The parameters W are modified during the training whereas the hyperparameters λ are constant during the training. So ANN training is a problem of learning whereas hyperparameter optimization is a problem of meta-learning: the goal is to learn to learn.

Usually there are several hyperparameters in an ANN. From now on in this thesis the term λ will refer to a set of hyperparameters, that is to say a vector of scalar hyperparameters.

2.2.2 Main issues

The Curse of Dimensionality

The *curse of dimensionality* is a phenomenon that appears when solving problems in high dimensional spaces. The problem is that the volume of a space grows exponentially with the dimension of the space. This makes the exploration of high dimension spaces difficult.

For instance, in order to sample evenly $[0, 1]^2$ with no more than a 0.1 distance between two points, $10^2 = 100$ points are required. In order to sample evenly $[0, 1]^{10}$ with no more than a 0.1 distance between two points, $10^{10} = 10$ billion points are required.

The Cost of Observations

In order to evaluate the performance of a set of hyperparameters an artificial neural network parametrized with these hyperparameters must be trained on a training set. This training is very time consuming since usually many layers with many parameters to trained are involved. Even with large computational resources, this training may take minutes or hours. Therefore, in order to be done in a reasonable amount of time, the optimization process has to be done using few observations.

The Score Function

Usually, the score function which is used to evaluate the performance of a hyperparameter is a metric like accuracy, F-score, AUC, etc [9]. These metrics are not differentiable with respect to the hyperparameters. Therefore, gradient based optimization methods like stochastic gradient (see 2.1.2) cannot be used.

2.3 Existing Solutions for Hyperparameter Optimization

When it comes to hyperparameter optimization, all approaches can be split into model-free and model-based. Model-based techniques build a model of the hyperparameter space during its exploration whereas model-free algorithms do not use the knowledge about the solution space during the optimization.

The figure below shows different approaches and their corresponding algorithms: grid search [10], random search [10], Tree Parzen Estimator [11] (refered as TPE), Spearmint [12], Radial Basis Functions (RBF) Surrogate Model [13], Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [14], Particle Swarm Optimization (PSO) [15].

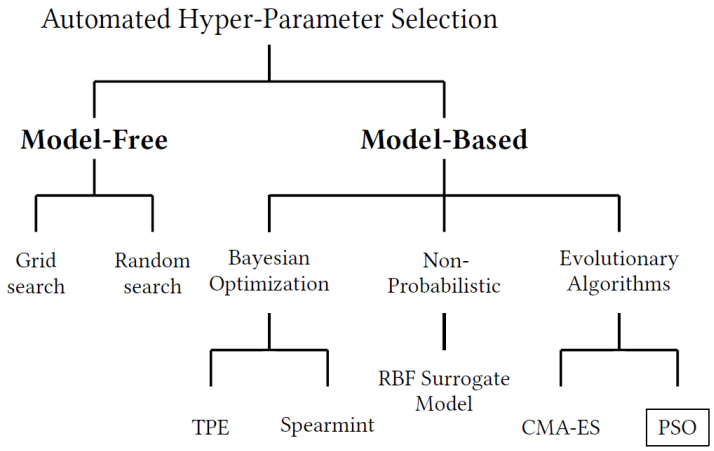


Figure 2.6: Approaches for the automated hyper-parameter selection [16]

This section presents two model-free solutions (grid search and random search) and two model based solutions (TPE, PSO).

It has been chosen to focus on these solutions in order to study very diverse approaches: naive solutions as a starting point (grid search/random search) and more complex solutions (TPE for a Bayesian approach, PSO for a non-Bayesian approach).

2.3.1 Model-free Algorithms

Grid Search [10]

A range of values is defined for each component of the hyperparameter and a grid is defined in the search space. Then the network is trained for every hyperparameter in the grid. A logarithmic scale can be used to cover a larger range of values.

Commonly several grid searches are sequentially performed. The user first performs a grid search with a large range of values. Then the user refines its search in the most promising area of search space.

Grid search is easy to implement however it suffers from the curse of the dimensionality. For a hyperparameter space of dimension k with n values tested for each dimension, n^k networks need to be trained. Therefore the complexity grows exponentially with the dimension. Grid search is reliable in low dimensional spaces [10].

Another practical issue with grid search is the lack of flexibility for the number of hyperparameters to be evaluated (it is n^k with k search space dimension and n an integer). For instance with $k = 5$, the number of hyperparameters which are evaluated can be $2^5 = 32$, $3^5 = 243$, $4^5 = 729$. No intermediate values like 50 or 100 can be used.

Random Search [10]

A range of values is defined for each component of the hyperparameter and n hyperparameter are randomly sampled in this range. Then the network is trained for every hyperparameter. The benefit is that n different values are tested for each dimension.

Therefore random search gives a better coverage than grid search. Bergstra and Bengio showed random search will have greater chance of finding effective values for each hyperparameter component [10](see figure 2.7). Like grid search random search is also easy to implement.

Figure 2.7 shows the results of Bergstra and Bengio's experiments [10]. They

created 100 search problems. For each problem a hyper-rectangle of dimension 5 (to simulate a problem of hyperparameter optimization in dimension 5) is designed and a volume representing 1% of the hyper-rectangle is sampled. The goal is to find this volume. Then for each problem up to 512 points are sampled in the hyper-rectangle using a search algorithm (grid search, random search, quasi-random search). If the 1% volume is reached, it is a success. The success rate is the rate of success over the 100 problems. One can see the success rate is much greater with random search and quasi-random search.

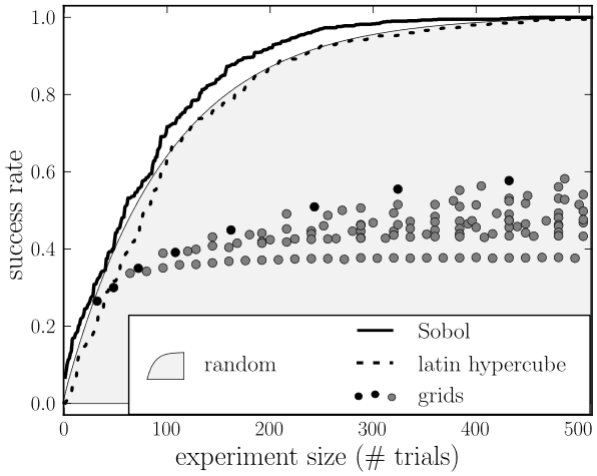


Figure 2.7: The efficiency in simulation of grid search, random search and quasi-random search [10]

If the number of samples is low, some areas of the search space may not be explored. This problem can be solved using quasi random sequences which are designed to cover the search space evenly. An example is latin hypercube: in order to sample n points from a d-dimensional space, each of the d dimensions are divided into n intervals and the sampling is done so that each of the n intervals is occupied by a point.

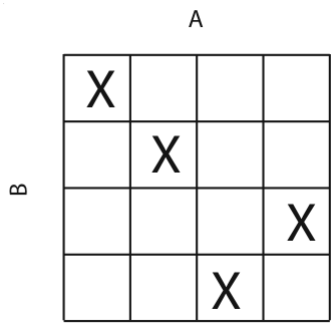


Figure 2.8: Example of latin hypercube sampling: each of the 4 rows and each of the 4 columns are occupied by a point

Sobol sequences are more complex solutions that also aim at covering the search space evenly.

2.3.2 Model-based Algorithms

Bayesian Optimization: Tree Parzen Estimator (TPE) [11]

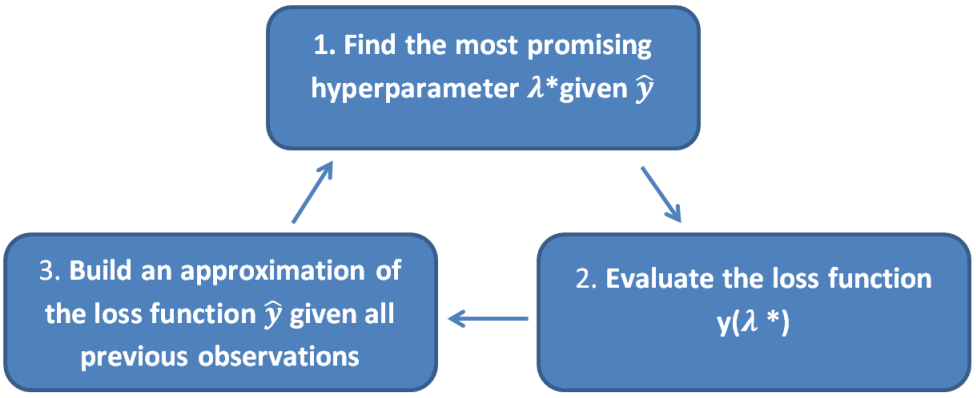
Sequential Model Based Optimization (SMBO)
Bayesian approaches uses Sequential Model Based Optimization (SMBO). SMBO tries to minimize validation loss **y** by sequentially selecting different hyperparameters using Bayesian reasoning.

The algorithm is composed of 3 main steps:

- step 1: given an approximation of the validation loss $\hat{\boldsymbol{y}}$, the most promising hyperparameter $\boldsymbol{\lambda}^*$ is computed. It will be the next guess.
  The meaning of *"the most promising"* is defined by an acquisition function **S** which tries to find a balance between exploring new areas in the search space and exploiting areas that are already known to have favourable values. It will be the next guess.
- step 2: the loss function is evaluated for the hyperparameter $\boldsymbol{\lambda}^*$. This is the most time-consuming step.

- step 3: given all the previous observations, the approximation of the loss function $\hat{\boldsymbol{y}}$ is updated.

This process is repeated until the maximum number of iterations is reached.



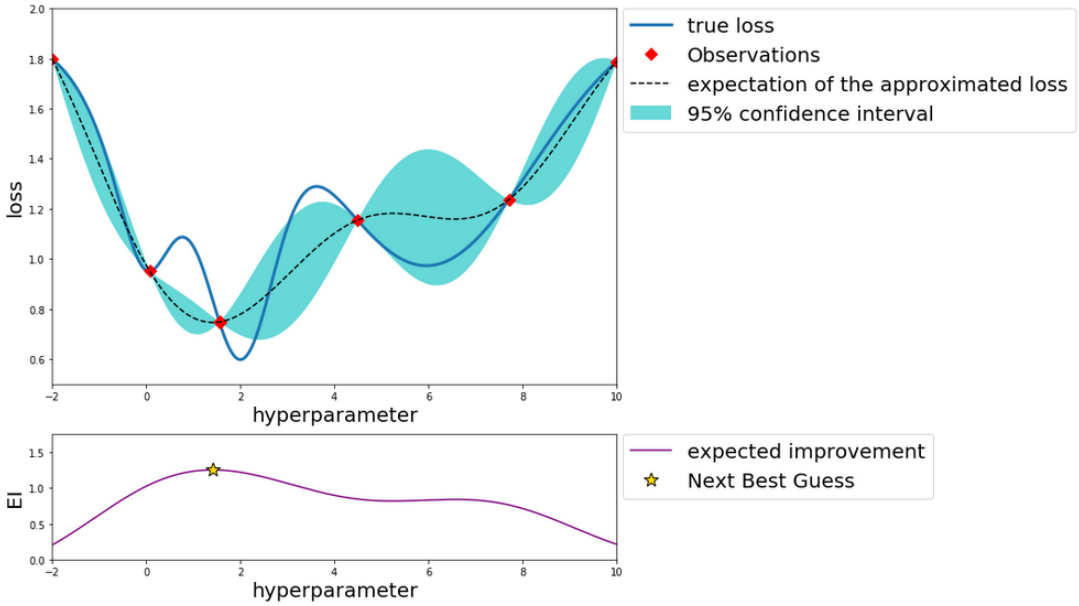
Algorithm: SMBO

Inputs: $\hat{y}_0$, the intial surrogate model
max_iter, the maximum number of iterations
S, the acquisition function
Result: λ_{best}, the best hyperparameter
$H \longleftarrow \emptyset$ # H is the history of observations
for $t \leftarrow 1$ **to** *max_iter* **do**
  $\lambda^* \longleftarrow \text{argmax}_\lambda \, S(\lambda, \hat{y}_{t-1})$ # step 1
  Evaluate $y(\lambda^*)$ # step 2
  $H \longleftarrow H \cup (\lambda^*, y(\lambda^*))$
  Fit a new model $\hat{y}_t$ to H # step 3
end
$\lambda_{best} = \text{argmax}_\lambda \, S(\lambda, \hat{y}_{max\_iter})$
return λ_{best}

Figure 2.9: SMBO algorithm

Example:
The goal is to find the minimum of the solid line. After 5 observations, a model $\hat{y}_5$ has been fitted to the data, the dashed line is the expectation of this

model. The acquisition function of this model is computed, the maximum of this acquisition function is the next point to be observed. After this observation, a new model $\hat{y}_6$ is computed and the process is repeated.



TPE

With TPE, the acquisition function is the Expected Improvement (EI):

$$EI_{y*}(\lambda) = \int_{-\infty}^{y*} (y^* - \hat{y}) p(\hat{y}|\lambda) d\hat{y} \tag{2.2}$$

where $\boldsymbol{\lambda}$ is a set of hyperparameter, $\hat{y}$ is a value of the validation loss, $y*$ is a value of the validation loss which is higher than the lowest observed. Intuitively, one can see the EI is high if both $(y^* - \hat{y})$ is high (which means great improvement) and $p(\hat{y}|\lambda)$ is high (which means likely improvement given the hyperparameter $\boldsymbol{\lambda}$).

TPE does not model $p(\hat{y}|\lambda)$ directly. Instead it uses Bayes rule: $p(\hat{y}|\lambda) = \frac{p(\lambda|\hat{y})p(\hat{y})}{p(\lambda)}$ where:

$$p(\lambda|\hat{y}) = \begin{cases} l(\lambda) & \text{if } \hat{y} < y* \\ g(\lambda) & \text{if } \hat{y} \geq y* \end{cases} \tag{2.3}$$

and:

$$p(\lambda) = \int_{-\infty}^{+\infty} p(\lambda|\hat{y}) p(\hat{y}) dy = \gamma l(\lambda) + (1-\gamma) g(\lambda) \text{ with } \gamma = p(\hat{y} < y*) \tag{2.4}$$

Therefore,

$$
\begin{aligned}
EI_{y*}(\lambda) &= \int_{-\infty}^{y*} (y^* - \hat{y}) p(\hat{y}|\lambda) d\hat{y} \\
&= \int_{-\infty}^{y*} (y^* - \hat{y}) \frac{p(\lambda|\hat{y})p(\hat{y})}{p(\lambda)} d\hat{y} \\
&= \frac{1}{p(\lambda)} \int_{-\infty}^{y*} (y^* - \hat{y}) p(\lambda|\hat{y}) p(\hat{y}) d\hat{y} \\
&= \frac{1}{p(\lambda)} l(\lambda) \int_{-\infty}^{y*} (y^* - \hat{y}) p(\hat{y}) d\hat{y} \\
&= \frac{1}{p(\lambda)} l(\lambda)\alpha, \;\; \alpha \text{ a constant w.r.t. } \lambda, \geq 0 \\
&= \frac{l(\lambda)\alpha}{\gamma l(\lambda) + (1-\gamma) g(\lambda)}, \;\; \alpha \text{ a constant w.r.t. } \lambda, \geq 0 \\
&= \frac{\alpha}{\gamma + (1-\gamma)\frac{g(\lambda)}{l(\lambda)}}, \;\; \alpha \text{ a constant w.r.t. } \lambda, \geq 0
\end{aligned}
\tag{2.5}
$$

So in order to maximize EI, the hyperparameters to be evaluated should have low probability under $\boldsymbol{g(\lambda)}$ and high probability under $\boldsymbol{l(\lambda)}$.

TPE in practice

First, a search space is defined for each hyperparameter component in order to define the range of the search and sample the first hyperparameters. Then hyperparameters are iteratively sampled using SMBO. These samples are divided into two groups based on the validation loss. By default the best 25% (which means $\gamma = 0.25$) go to the "good group" and and the other go the "bad group".

The two densities $\boldsymbol{l(\lambda)}$ and $\boldsymbol{g(\lambda)}$, respectively for the "bad group" and the "good group" are built: each sample defines a Gaussian distribution and all the Gaussian distributions of a group are stacked.

The hyperparameter that minimizes $\frac{g(\lambda)}{l(\lambda)}$ is the next guess and the two densities $l(\lambda)$ and $g(\lambda)$ are updated until the maximum number of iterations is reached.

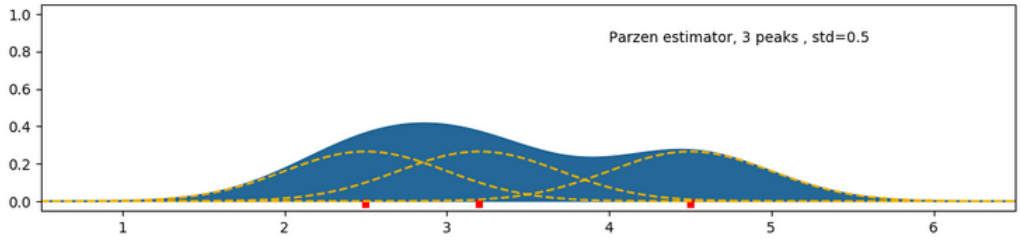


Figure 2.10: Construction of a density based on 3 samples

Evolutionary Algorithm: Particle Swarm Optimization (PSO) [17][18]

Particle Swarm Optimization (PSO) is a population-based optimization method. It was created by Kennedy, Eberhart and Shi [18][17] and it was inspired from animal social groups like herds, schools and flocks. The swarm is composed by particles which have a position in the search space. Particles move in the search space and cooperate according to simple mathematical formulas in order to find an optimal solution.

In the case of hyperparameter optimization, a particle's coordinate corresponds to a hyperparameter in the hyperparameter search space.

In a D-dimensional search space:

- the position of the i-th particle at time *t* is $\lambda_i(t) = (\lambda_{i1}(t), ..., \lambda_{iD}(t))$
- the best position so far of the i-th particle at time *t* is $pbest_i(t) = (pbest_{i,1}(t), ..., pbest_{i,D}(t))$
- the best position so far of the whole swarm at time *t* is $gbest(t) = (gbest_1(t), ..., gbest_D(t))$

First, the positions of the particles are randomly initialized in the search space and then the paryicles move in the search space. At each iteration, the position of the i-th particle is computed as follows:

$$\lambda_i(t+1) = \lambda_i(t) + v_i(t) \tag{2.6}$$

$$v_i(t+1) = \omega v_i(t) + c_1 r_1 (pbest_i(t) - \lambda_i(t)) + c_2 r_2 (gbest_i(t) - \lambda_i(t)) \tag{2.7}$$

$v_i(t)$ is the velocity of the i-th particle at time t, ω is an inertia weight scaling the previous time step velocity, c_1 and c_2 are two acceleration coefficients that scale the influence of the best personal position of the particle $pbest_i(t)$ and the best global position $gbest(t)$ and r_1 and r_2 are random variables within the range of 0 and 1.

The first term of the velocity $\omega v_i(t)$ in an inertia term that creates a balance between the previous movement and the changes of direction caused by new information.
The second term $c_1 r_1(pbest_i(t) - \lambda_i(t))$ is the "cognition" part which represents the private thinking of the particle itself.
The third term $c_2 r_2(gbest_i(t) - \lambda_i(t))$ is the "social" part which represents the collaboration among the particles.

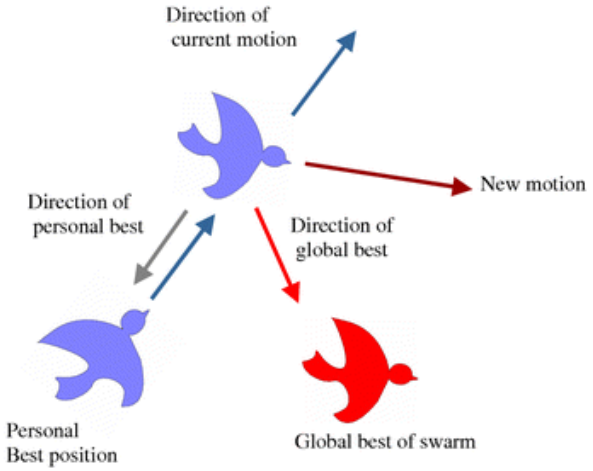


Figure 2.11: Representation of the update of a particle velocity [19]

Chapter 3

Methods

Chapter 3 presents the methodology that is used to evaluate and compare the hyperparameter optimization algorithms. First, the MNIST dataset, which is used to evaluate the algorithms, is introduced. Then, three experiments of hyperparameter optimization are presented. Finally the metrics that will be used for the results analysis are defined.

3.1 Data and Task

The goal of this thesis is to present and experiment some hyperparameter optimization algorithms. In order to obtain comparable results and draw conclusions that may be generalized, it has been chosen to test these algorithms on a widely used dataset: MNIST dataset [20].

The MNIST (Modified National Institute of Standards and Technology) dataset is composed of 70000 images of handwritten digits (0 to 9). Each image is a 28×28 pixels gray-scale image. The dataset is divided into a training set (60000 samples) and a test set (10000 samples).

Given a 28×28 input image, the task is to predict the corresponding digit.

Figure 3.1: Examples of MNIST images [20]

3.2 Starting Point: LeNet-1

The goal of this thesis is not to study complex convolutional neural networks and the computational resources provided for this thesis were limited. Therefore a rather simple network has been chosen as a starting point: LeNet-1 [21]. Created in 1995 by LeCun et al., LeNet-1 was one of the first CNN tested on MNIST dataset. It showed good results (1.7% error rate) although it is not state of the art now (0.23 % error rate has been reached [22]).

By default, the network is composed of 7 layers:

	Layer type	# conv. filters	Kernel size	Padding	Activation	Number of dense units
1	Input	-	-	-	-	-
2	Conv2D	4	5×5	no padding	tanh	-
3	Average Pooling2D	-	2×2	-	-	-
4	Conv2D	12	5×5	no padding	tanh	-
5	Average Pooling2D	-	2×2	-	-	-
6	Flatten	-	-	-	-	-
7	Dense	-	-	-	sigmoid	10

Table 3.1: Detailed architecture of LeNet-1

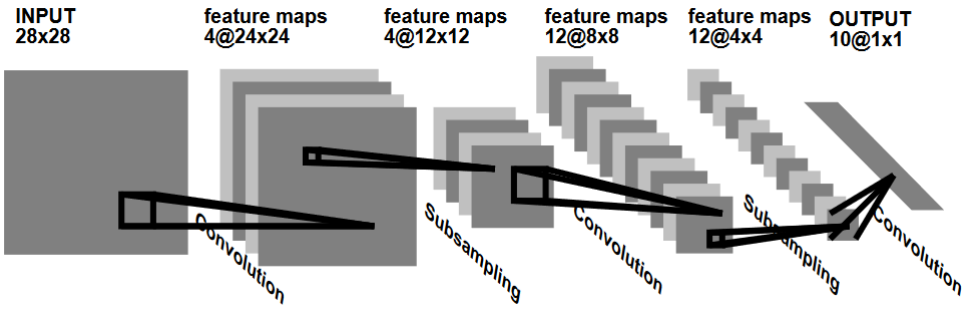


Figure 3.2: General Structure of LeNet-1 [21]

3.3 Experiments

3.3.1 Experiment 1: Optimizing LeNet-1 with Variable Complexity

Motivation

In this experiment, the overall structure of LeNet-1 (two convolutional layers and one dense layer) is fixed but the number of parameters of each layer can vary. The goal of this task is to find the network complexity that achieves the best results given a fixed architecture.

Network

For this experiment, no additional layer is added to LeNet-1 but the size of the two convolutional layers are not fixed. Four hyperparameters can vary:

- $nconv1$: the number of convolutional filters in the first convolutional layer (integer ranging from 1 to 100).
- $size\_conv1$: the width of the convolutional filters in the first convolutional layer (integer ranging from 2 to 8). The convolutional filters are assumed to be squared, thus the kernel size is $size\_conv1 \times size\_conv1$

- $nconv2$: the number of convolutional filters in the second convolutional layer (integer ranging from 1 to 100).
- $size\_conv2$: the width of the convolutional filters in the second convolutional layer (integer ranging from 2 to 8). The convolutional filters are assumed to be squared, thus the kernel size is $size\_conv2 \times size\_conv2$.

	Layer type	# conv. filters	Kernel size	Padding	Activation	Number of dense units
1	Input	-	-	-	-	-
2	Conv2D	**nconv1**	**size_conv1** $\times$ **size_conv1**	no padding	tanh	-
3	Average Pooling2D	-	2×2	-	-	-
4	Conv2D	**nconv2**	**size_conv2** $\times$ **size_conv2**	no padding	tanh	-
5	Average Pooling2D	-	2×2	-	-	-
6	Flatten	-	-	-	-	-
7	Dense	-	-	-	sigmoid	10

Table 3.2: Detailed Architecture of LeNet-1_exp1 (variable hyperparameters are in bold)

Training

The network is implemented using *Keras* library [23], the loss function is the *categorical crossentropy*, the optimizer is *Adam* optimizer [6] with a learning rate equal to 0.0001 (and other parameters set to default [6]). The batch size is set to 32. The metrics to evaluate the network is the accuracy.

3.3.2 Experiment 2: Optimizing LeNet-1 with Variable Regularization Capacity

Motivation

In experiment 2, the overall structure of LeNet-1 (the convolutional layers and the dense layer) is fixed. However three changes are made to improve the generalization capacity of the network: the use of dropout, the use of weight decay (L2-regularization) and a variable batch size for the training.

Network

Three hyperparameters can vary:

- $l2\_reg$: the coefficient β that weighs the L2-regularization (see 2.1.2). It is a float ranging from 1 to 10^{-10}.
- $dropout\_rate$: the dropout rate (see 2.1.2). It is a float ranging from 0 to 1.
- $batch\_size$: the size of the mini-batches (see 2.1.2). It is an integer ranging from 1 to 100.

	Layer type	# conv. filters	Kernel size	Padding	Activation	Number of dense units	dropout rate	weight decay
1	Input	-	-	-	-	-	-	-
2	Conv2D	4	5×5	no padding	tanh	-	-	-
3	Average Pooling2D	-	2×2	-	-	-	-	-
4	Conv2D	12	5×5	no padding	tanh	-	-	-
5	Average Pooling2D	-	2×2	-	-	-	-	-
6	Flatten	-	-	-	-	-	-	-
7	Dropout	-	-	-	-	-	**dropout_rate**	-
8	Dense	-	-	-	sigmoid	10	-	**l2_reg**

Table 3.3: Detailed architectures of LeNet-1_exp2 (variable hyperparameters are in bold)

Training

The network is implemented using *Keras* library, the loss function is the *categorical crossentropy*, the optimizer is *Adam* optimizer with a learning rate equal to 0.0001. The batch size **batch_size** is a variable. The metrics to evaluate the network is the accuracy.

3.3.3 Experiment 3: Optimizing "augmented LeNet-1" with Variable Regularization Capacity

Motivation

In order to test the scalability of hyperparameter optimization, a network with more hyperparameters has been studied. Another reason is that optimization algorithms also include hyperparameters. For instance PSO includes 4 hyperparameters (the number of particles, the three coefficients that weigh inertia, individual behaviour, collective behaviour). TPE also includes hyperparameters, such as the quantile γ (see 2.3.2) for instance. As a consequence, in order to actually reduce the complexity of the problem, these algorithms have to be effective to optimize more hyperparameters that they have themselves.

The network is composed of 2 convolutional layers, two dense layers but also dropout and batch normalization and weight decay. Therefore there are more hyperparameters. The goal is to test if the optimization algorithms perform well when the dimensionality of the search space increases.

This network is refered as "augmented LeNet-1".

Network

Eight hyperparameters can vary (see the architecture 3.3.3):

- $dp1$, $dp2$, $dp3$, $dp4$: the four dropout rates. It is a float ranging from 0 to 1.
- $l2\_reg1$, $l2\_reg2$: the coefficients β that weighs the two L2-regularization. It is a float ranging from 1 to 10^{-10}.
- lr: the learning rate. It is a float ranging from 10^{-1} to 10^{-6}. This range has been chosen since learning rates higher than 10^{-1} lead to unstable learning and learning rates lower than 10^{-6} lead to a very slow convergence.
- $batch\_size$: the size of the mini-batches. It is an integer ranging from 16 to 512. This range has been chosen since batch sizes lower than 16

lead to a very long training whereas batch sizes higher than 512 lead to memory issues.

	Layer type	# conv. filters	Kernel size	Padding	Activation	Number of dense units	dropout rate	weight decay
1	Input	-	-	-	-	-	-	-
2	BatchNorm	-	-	-	-	-	-	-
3	Conv2D	32	3×3	no	relu	-	-	-
4	Dropout	-	-	-	-	-	**dp1**	-
4	MaxPooling2D	-	2×2	-	-	-	-	-
5	BatchNorm	-	-	-	-	-	-	-
6	Conv2D	64	3×3	no	relu	-	-	-
7	Dropout	-	-	-	-	-	**dp2**	-
8	MaxPooling2D	-	2×2	-	-	-	-	-
9	Flatten	-	-	-	-	-	-	-
10	BatchNorm	-	-	-	-	-	-	-
11	Dropout	-	-	-	-	-	**dp3**	-
12	Dense	-	-	-	relu	150	-	**l2reg-1**
13	BatchNorm	-	-	-	-	-	-	-
14	Dropout	-	-	-	-	-	**dp4**	-
15	Dense	-	-	-	sigmoid	10	-	**l2reg-2**

Table 3.4: Detailed architectures of "augmented LeNet-1" (variable hyperparameters are in bold)

Training

The network is implemented using *Keras* library, the loss function is the *categorical crossentropy*, the optimizer is *Adam* optimizer with a variable learning rate equal **lr**. The batch size **batch_size** is also variable. The metrics to evaluate the network is the accuracy.

3.4 Hyperparameter Optimization

3.4.1 Normalization

The hyperparameters may vary in very different ranges and be of different data types (float, integer). To create a generic method, all the hyperparameters are associated to a float ranging from 0 to 1.

For experiment 1,
$[\lambda_1, \lambda_2, \lambda_3, \lambda_4] \longrightarrow [nconv1, sizeconv1, nconv2, sizeconv2]$
such that:

- $nconv1 = 1 + round(99\lambda_1)$
- $sizeconv1 = 2 + round(6\lambda_2)$
- $nconv2 = 1 + round(99\lambda_3)$
- $sizeconv2 = 2 + round(6\lambda_4)$

So the initial hyperparameter optimization problem becomes a problem of optimization in $[0, 1]^4$.

For experiment 2,
$[\lambda_1, \lambda_2, \lambda_3] \longrightarrow [l2\_reg, dropout\_rate, batch\_size]$ such that:

- $l2\_reg = 10^{(-10\lambda_1)}$
- $dropout\_rate = \lambda_2$
- $batch\_size = 1 + round(99\lambda_3)$

So the initial hyperparameter optimization problem becomes a problem of optimization in $[0, 1]^3$.

For experiment 3,
$[\lambda_1, ..., \lambda_8] \longrightarrow [dp1, dp2, dp3, dp4, l2\_reg1, l2\_reg2, lr, batch\_size]$ such that:

- $dp1 = \lambda_1$
- $dp2 = \lambda_2$
- $dp3 = \lambda_3$
- $dp4 = \lambda_4$
- $l2\_reg1 = 10^{(-10\lambda_5)}$
- $l2\_reg2 = 10^{(-10\lambda_6)}$
- $lr = 10^{-(1+5\lambda_7)}$
- $batch\_size = 16 + round(496\lambda_8)$

So the initial hyperparameter optimization problem becomes a problem of optimization in $[0, 1]^8$.

3.4.2 Optimization

For the three experiments, random search, TPE and PSO are used to optimize the hyperparameters.

The algorithms for random search and PSO were implemented using Python whereas the library *hyperopt* [24] was used for TPE. The convolutional neural networks were implemented and trained using *Keras* [23] and a NVIDIA GeForce GTX 1080.

For experiments 1 and 2, which take place in low dimension, 50 hyperparameters are evaluated. Random search consists in randomly sampling 50 hyperparameters. In PSO, 5 particles are randomly initialized in the search space and 10 iterations are done, so 50 hyperparameters are evaluated. With TPE, before the algorithm runs, the first 5 hyperparameters to be evaluated are the same as PSO initialization, so PSO and TPE can be compared with same initial conditions.

For experiment 3, which takes place in higher dimension, 100 hyperparameters are evaluated. Random search consists in randomly sampling 100 hyperparameters. In PSO, 10 particles are randomly initialized in the search space and 10 iterations are done, so 100 hyperparameters are evaluated. With TPE, 100 hyperparameters are evaluated as well with the same initialization as PSO.

3.5 Metrics

The goal of hyperparameter optimization is to explore the most promising areas of the search space without prior knowledge. The evaluation of the algorithms and the analysis of the results will be based upon two capabilities: exploration and exploitation. Metrics that quantify these capabilities were designed and are contributions from this thesis.

3.5.1 Exploration

Exploration is testing large portions of the search space with the hope of finding promising solutions. It is a strategy of diversification.

Metric 1: Dispersion

The dispersion is defined as the standard deviation of the coordinates of the hyperparameters in the hyperparameter search space.

Metric 2: Number of Intervals Explored

For each dimension of the search space, the search space $[0, 1]$ is divided into two intervals $[0, 0.5]$ and $[0.5, 1]$. This division is applied for each dimension, so for a d-dimensions space, there are 2^d intervals.

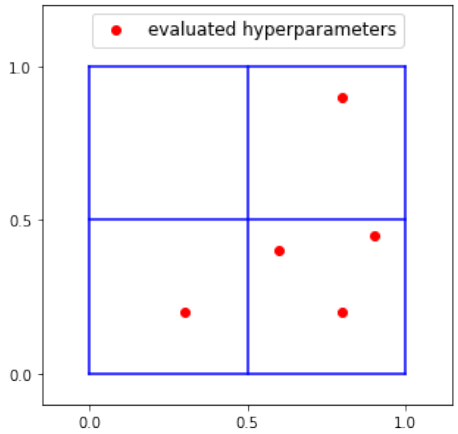


Figure 3.3: Example of the "Number of Intervals Explored" in 2D ($[0, 1]^2$). The 5 hyperparameters explored 3 intervals out of 4.

3.5.2 Exploitation

Exploitation is testing limited portions of the search space with the hope of improving a promising solution that we already have tested. It is a strategy of intensification.

Metric: Mean of the Evaluations

This is the mean of the error rate (in %) for all the hyperparameters settings evaluated during the optimization process.

Chapter 4

Results

4.1 Exploration

4.1.1 Metric 1: Dispersion

Experiment \ Algorithm	random search	TPE	PSO
1 (optimization in $[0,1]^4$)	0.284	0.239	0.184
2 (optimization in $[0,1]^3$)	0.284	0.282	0.183
3 (optimization in $[0,1]^8$)	0.285	0.251	0.155

Table 4.1: Dispersion for the 3 experiments and the 3 optimization algorithms.

4.1.2 Metric 2: Number of Sub-intervals Explored

Experiment \ Algorithm	random search	TPE	PSO
1 (optimization in $[0,1]^4$, ie 16 intervals)	15	15	9
2 (optimization in $[0,1]^3$, ie 8 intervals)	8	8	8
3 (optimization in $[0,1]^8$, ie 256 intervals)	83	64	28

Table 4.2: Number of intervals Explored for the 3 experiments and the 3 optimization algorithms.

4.1.3 Analysis

Unsurprisingly it is the random search that focuses more on exploration for the experiments that were conducted. It is interesting to notice that TPE uses more a strategy of exploration than PSO. In low-dimension search spaces, PSO has a good exploration performance (in experiment 2: all the 8 subintervals are exlored, like TPE and random search). In high-dimension search spaces, in term of number of subintervals explored, TPE is closer to random search than to PSO. PSO explored a very limited part of search space (28 out of 256 subintervals, ie 11%).

4.2 Exploitation

4.2.1 Metric: Mean of the Evaluations

Experiment \ Algorithm	random search	TPE	PSO
1 (optimization in $[0,1]^4$)	1.20	1.01	0.95
2 (optimization in $[0,1]^3$)	2.8	2.3	1.3
3 (optimization in $[0,1]^8$)	25.2	10.5	2.2

Table 4.3: Mean of the Evaluations for the 3 experiments and the 3 optimization algorithms

It is also interesting to look at the distribution of the error rates (see figure 4.2). The top line of the boxes is the first quartile, the orange line is the median, the bottom line is the third quartile.

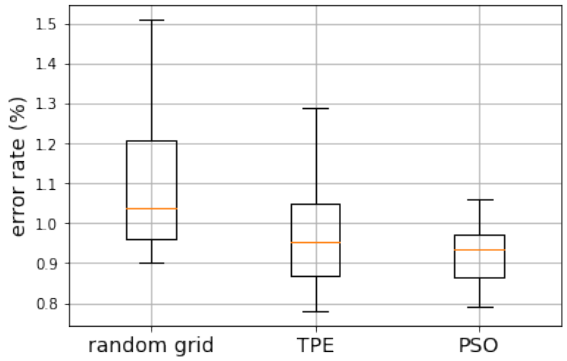


Figure 4.1: Distribution of the evaluations for experiment 1.

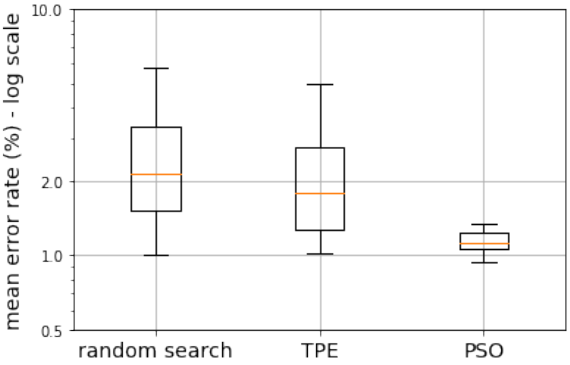


Figure 4.2: Distribution of the evaluations for experiment 2.

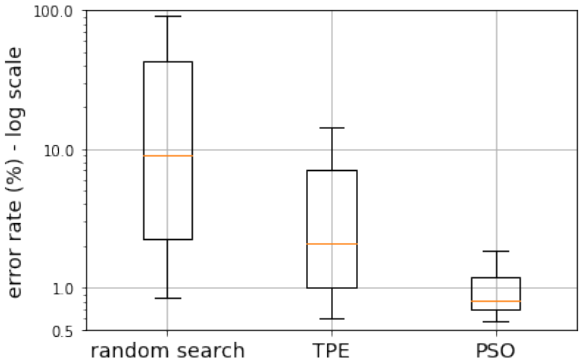


Figure 4.3: Distribution of the evaluations for experiment 3.

4.2.2 Analysis

For both experiments, PSO is the algorithm that shows best performance in term of exploitation since most of the hyperparameters it evaluated have low error rates. For instance looking at the boxplots of experiment 3, the 95% best error rates of PSO are equivalent to the 50% best error rates of TPE and to the 25% best error rates of random search.

4.3 Exploration/Exploitation Tradeoff

Exploration and exploitation are antagonist notions. A strategy of exploration leads to observing very diverse areas of the search space and therefore not really using the already exploited areas. A strategy of exploitation leads to focusing on areas of the search that are known to be good, and therefore not really exploring the search space. So a tradeoff must be found between exploration and exploitation.

In practical, the purpose of hyperparameter optimization is obtain the best performance possible with the fewest observations possible. The figures below shows the evolution of the lowest error reached so far during the optimization process.

The following plots show the evolution of the lowest error rate reached during the optimization process. For instance let's take a point whose coordinates are 'number of observations' $= 10$, and 'lowest error rate reached' $= 0.79$. This means after having evaluated 10 hyperparameters, the best hyperparameter (out of the 10 hyperparameters), has an error rate of 0.79%.

For PSO, the 'lowest error rate reached' is updated after each particle's update, and not after a complete swarm update.

4.3.1 Experiment 1

optimization algorithm	random search	TPE	PSO
number of observations to reach an error rate of 1.0 (%)	6	8	6
number of observations to reach an error rate of 0.9 (%)	23	8	8
number of observations to reach an error rate of 0.8 (%)	not reached	43	8
lowest error rate after 50 observations (%)	0.90	0.78	0.79

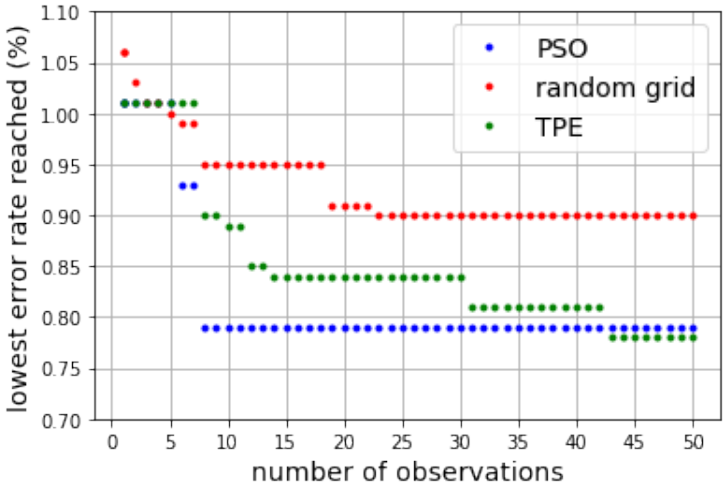


Figure 4.4: Evolution of the lowest error rate for experiment 1.

4.3.2 Experiment 2

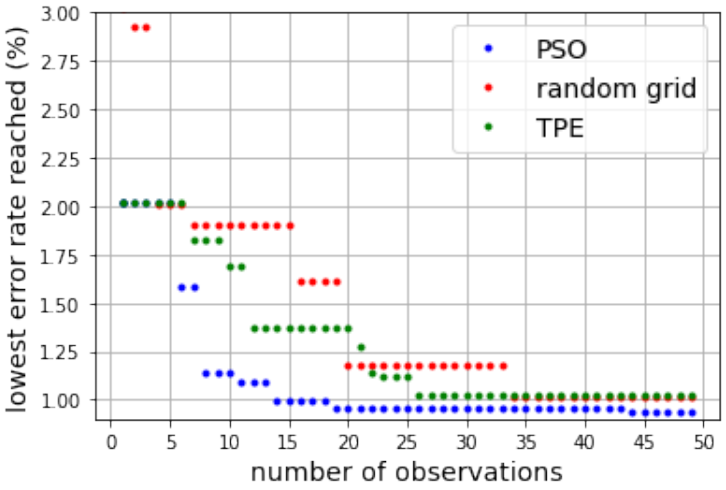


Figure 4.5: Evolution of the lowest error rate for experiment 2.

optimization algorithm	random search	TPE	PSO
number of observations to reach an error rate of 2.0 (%)	7	7	6
number of observations to reach an error rate of 1.5 (%)	20	12	8
number of observations to reach an error rate of 1.0 (%)	not reached	not reached	14
lowest error rate after 50 observations (%)	1.01	1.02	0.96

4.3.3 Experiment 3

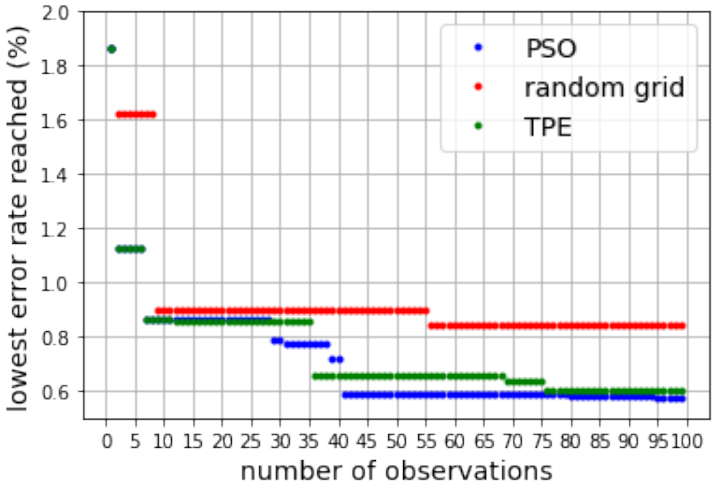


Figure 4.6: Evolution of the lowest error rate for experiment 3.

optimization algorithm	random search	TPE	PSO
number of observations to reach an error rate of 1.0 (%)	9	7	7
number of observations to reach an error rate of 0.8 (%)	not reached	36	29
number of observations to reach an error rate of 0.6 (%)	not reached	76	41
lowest error rate after 100 observations (%)	0.84	0.60	0.57

4.3.4 Analysis

In low dimension, the final score obtained after the whole optimization process are close for random search, TPE and PSO: there is a 0.05 difference between the worse final score (random search) and the best final score (PSO). However PSO is much faster to converge towards good scores. TPE is also quicker than random search.

In high dimension, the lack of exploitation capacity seems to prevent random search from converging towards good scores like PSO and TPE. TPE and PSO converge towards similar scores but like in low dimension, PSO is a bit faster to converge.

Chapter 5

Application to Audio Classification

Audio processing is the one of the main subjects of research business of the team where this thesis was conducted (team *Ambient Intelligence* of *Orange Labs Lannion*). Chapter 5 presents a real-world application of hyperparameter optimization. The algorithms detailed and evaluated in Chapter 2, 3 and 4 are applied to a task of audio tagging. This problem is made similar as the task of image classification by using mel-spectrograms, a 2D representation of sounds.

5.1 Data and Task

The dataset is composed of 2351 recordings in the training dataset and 443 recordings in the validation dataset. Each recording is a 10-seconds audio segment recorded in the streets of New York City.

8 labels can be present in the audio recordings:

- engine
- machinery impact
- non-machinery impact
- powered saw
- alert signal
- music

- human voice
- dog barking

Labels are non-exclusive: several (or none) classes can be present in each recording. For each recording, the goal is to output a probability of presence for each class. The performance of the model is measured using micro-averaged AUPRC (Area Under Precision Recall Curve [25]).

5.2 Feature Engineering

5.2.1 Definition of Melspectrograms

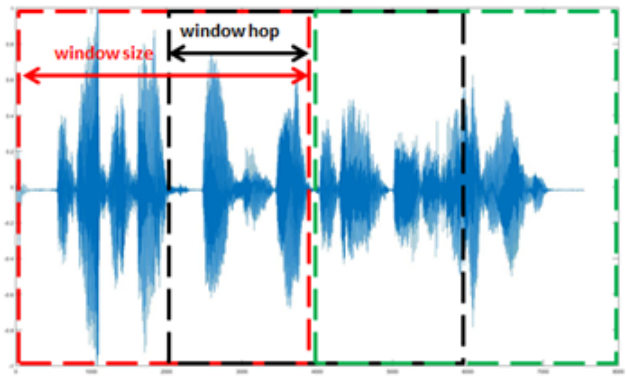
Since artificial neural networks have showed good performance on image classification tasks, audio inputs are often converted into images through mel-spectograms. A spectrogram is a visual representation of a sound that contains information both in time domain and in frequency domain. It represents the evolution of the frequency spectrum over time.

First the original audio signal is decomposed into signals of length *window_size*. The overlap between two consecutive signals is the *window_hop* (1 in figure 5.1). The usual size of *window_size* and *window_hop* is a few milliseconds.

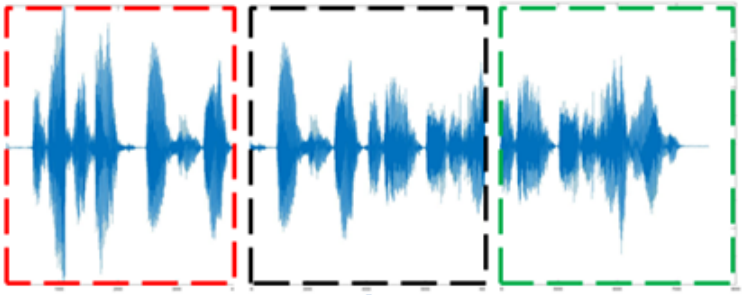
Then for each window the Fourier Transform magnitude is computed (2 in figure 5.1).

Finally the Fourier Transform magnitudes are stacked: the horizontal axis corresponds to the time domain and the vertical axis corresponds to the frequency domain (3 in figure 5.1).

A mel-spectrogram is a spectrogram where a mel scale is used in the frequency domain. A mel scale is a piece-wise logarithmic scale based on human perception of sounds. There does not exist a single definition of mel scale. In this project, the *melspectrogram* function of the *librosa*[26] library has been used.



(1)



(2)

Frequency	FT magnitude window 1	FT magnitude window 2	FT magnitude window 3
0	0.256	0.964	0.547
1	0.158	0.234	0.368
2	0.129	0.368	0.256
---	---	---	---

(3)

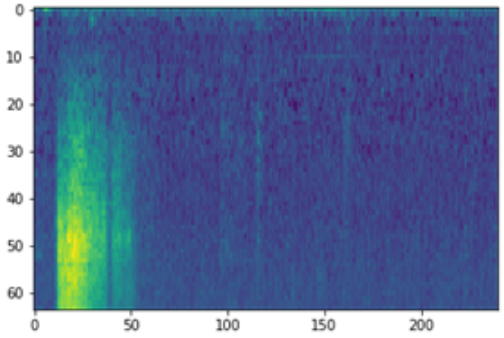


Figure 5.1: Construction of a spectrogram

5.2.2 Feature Engineering for the Audio Classification Task

First the recordings are re-sampled using a sampling rate of 22050Hz, which enables not to lose information from frequencies lower than 11025Hz and therefore keep most of the information from the signal. Then three features are extracted from these signals:

- The mel-spectrograms using 64 mel-bands and a hop length of 512 thus resulting a 64 rows x 431 columns image.
- The averaged value of the harmonic and percussive components (64 rows x 431 columns image).
- The derivative of the mel spectrograms (64 rows x 431 columns image).

These spectrograms have been extracted using the *librosa* library [27]. The figure below represents the three spectrograms extracted from an alert-signal recording.

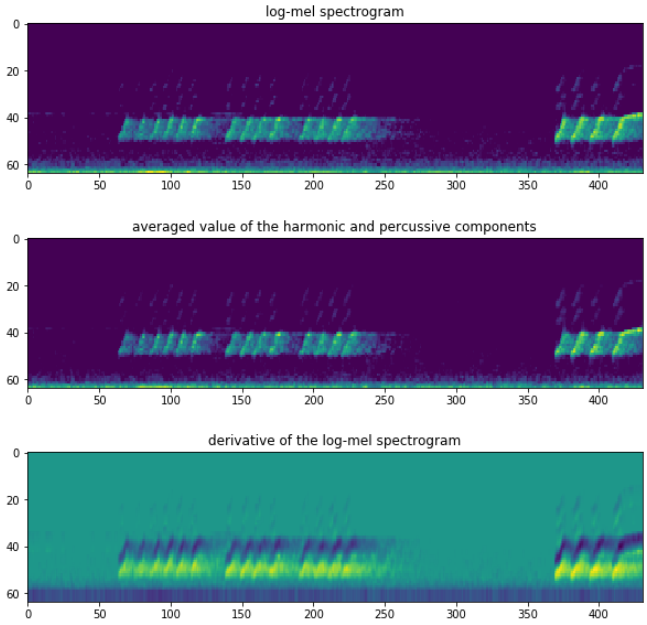


Figure 5.2: The three input images corresponding to a recording

5.3 Model

5.3.1 Model structure

A VGG-style [28] convolutional neural network is used to detect the classes from the input spectrograms:

Input 64 x 431 x 3
64 x conv 3x3 64 x conv 3x3
MaxPooling 2x2
128 x conv 3x3 128 x conv 3x3
MaxPooling 2x2
256 x conv 3x3 256 x conv 3x3 256 x conv 3x3
MaxPooling 2x2
512 x conv 3x3 512 x conv 3x3 512 x conv 3x3
MaxPooling 2x2
512 x conv 3x3 512 x conv 3x3 512 x conv 3x3
MaxPooling 2x2
Flatten
1024-Fully Connected + L2-regularization ReLu Activation Dropout
1024-Fully Connected + L2-regularization ReLu Activation Dropout
512-Fully Connected + L2-regularization ReLu Activation Dropout
512-Fully Connected + L2-regularization ReLu Activation Dropout
8-Fully Connected + L2-regularization Sigmoid Activation

All convolutional layers are initialized with the weights of VGG16 pre-trained on Imagenet dataset [29] but they remain unfrozen during the training. This

model has 30,188,360 parameters.

5.3.2 Data Augmentation

The training set is quite small (2351 samples). Data augmentation is a way to artificially increase the size of the training set and avoid overfitting. Mixup is a data augmentation method that has been experimented for this task. A new sample is created by linearly combining two samples. This linear combination is applied to both the melspectrograms and the corresponding labels. From two samples $\{input : x_1, target : y_1\}$ and $\{input : x_2, target : y_2\}$ a 'new' sample is created: $\{input : x_3 = \lambda x_1 + (1 - \lambda)x_2, target : y_3 = \lambda y_1 + (1 - \lambda)y_2\}$ where $\lambda \sim \beta(mixup\_rate)$ [30].

5.3.3 Model Training

binary_crossentropy is used as a loss function which is optimized using *Adam* optimizer with a learning rate of 0.00001 during 100 epochs.

The model is trained using a GPU (NVIDIA GeForce GTX 1080), the training takes about 1 hour.

5.4 Optimization of the Model

Four hyperparameters can vary:

- the $dropout\_rate$ of all the layers that use dropout
- $l2\_reg$, the L2-regularization constant of all the layers that use L2-regularization
- $batch\_size$, the size of the mini-batches
- the $mixup\_rate$

For this task, the training is pretty long (1 hour) so only 30 hyperparameters are evaluated. The three algorithms presented earlier are evaluated: random search, PSO (using 5 particles) and TPE.

5.5 Results

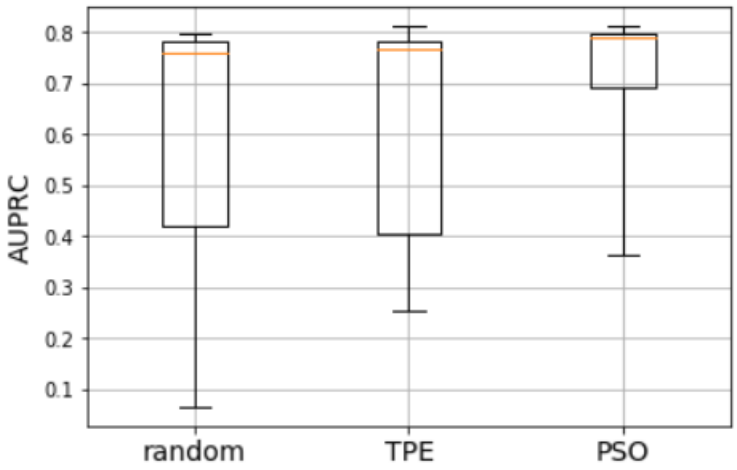


Figure 5.3: Distribution of the AUPRC for the three algorithms

.	random search	TPE	PSO
max AUPRC	0.798	0.811	0.812
mean AUPRC	0.597	0.617	0.707
number of observations to reach a 0.8 AUPRC	not reached	22	21

Table 5.1: Performance of the optimization processes

PSO explored most interesting areas of the search space (mean AUPRC is higher for PSO than for random search and TPE). However, looking at the best score and the speed of convergence, TPE and PSO have very similar results. It is interesting to notice that the optimization algorithms show similar results on this audio classification task than on the MNIST dataset.

Chapter 6

Conclusion

Hyperparameter optimization is a really difficult problem for two main reasons: the curse of the dimensionality and the technical necessity to optimize a score function with very few observations.

In this thesis, three methods (random search, TPE, PSO) were presented and evaluated. Metrics were also defined in order to quantify the efficiency of these methods in term of search space exploring. Three different methods were conducted on MNIST dataset to test the algorithms on diverse tasks. Unsurprisingly, model based solutions like TPE and PSO performs better than model free algorithms. PSO focuses more on exploitation whereas TPE focuses more on exploitation. Quantitative evidence of these statements can be found in the results chapter.

Since the optimization must be done evaluating few hyperparameters, exploitation is really important. This is probably the reason why PSO converges little quicker than PSO.

Another optimization method that would have been interesting to evaluate is manual tuning. Indeed, even if PSO showed to be more effective than TPE and random search, it has not been proven that PSO outperforms human at hyperparameter tuning.

There is still a lot to do in the field of hyperparameter optimization. For instance creating a balance between exploration and exploitation by varying the hyper-hyperparameters (hyperparameters of the optimization algorithms)

or studying the sensitivity of the optimization algorithms to their own hyperparameters.

Bibliography

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

[2] *An intuitive guide to Convolutional Neural Networks*. https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/.

[3] *Introduction to Convolutional Neural Networks for Vision Tasks*. https://pythonmachinelearning.pro/introduction-to-convolutional-neural-networks-for-vision-tasks/.

[4] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 448–456. URL: http://dl.acm.org/citation.cfm?id=3045118.3045167.

[5] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299.

[6] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[7] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[8] Marc Claesen et al. "Easy Hyperparameter Search Using Optunity". In: (Dec. 2014).

[9] https://scikit-learn.org/stable/modules/model_evaluation.html.

[10] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305. ISSN: ISSN 1533-7928. URL: http://www.jmlr.org/papers/v13/bergstra12a.html.

[11] James S. Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-Taylor et al. Curran Associates, Inc., 2011, pp. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[12] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 2951–2959. URL: http://dl.acm.org/citation.cfm?id=2999325.2999464.

[13] Ilija Ilievski et al. "Hyperparameter Optimization of Deep Neural Networks Using Non-Probabilistic RBF Surrogate Model". In: (July 2016).

[14] Nikolaus Hansen and Andreas Ostermeier. "Completely Derandomized Self-Adaptation in Evolution Strategies". In: *Evol. Comput.* 9.2 (June 2001), pp. 159–195. ISSN: 1063-6560. DOI: 10.1162/106365601750190398. URL: http://dx.doi.org/10.1162/106365601750190398.

[15] Yuhui Shi and Eberhart RC. "A Modified Particle Swarm Optimizer". In: vol. 6. June 1998, pp. 69–73. ISBN: 0-7803-4869-9. DOI: 10.1109/ICEC.1998.699146.

[16] Pablo Ribalta Lorenzo et al. "Particle Swarm Optimization for Hyper-parameter Selection in Deep Neural Networks". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Berlin, Germany: ACM, 2017, pp. 481–488. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071208. URL: http://doi.acm.org/10.1145/3071178.3071208.

[17] Yuhui Shi and Russell C. Eberhart. "A Modified Particle Swarm Optimizer". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. Washington, DC, USA: IEEE Computer

Society, May 1998, pp. 69–73. DOI: 10.1109/ICEC.1998.699146.

[18] James Kennedy and Russell C. Eberhart. "Particle swarm optimization". In: *Proceedings of the IEEE International Conference on Neural Networks*. 1995, pp. 1942–1948.

[19] Abbas Ahmadi, Fakhri Karray, and Mohamed S. Kamel. "Flocking based approach for data clustering". In: *Natural Computing* 9.3 (Sept. 2010), pp. 767–791. ISSN: 1572-9796. DOI: 10.1007/s11047-009-9173-5. URL: https://doi.org/10.1007/s11047-009-9173-5.

[20] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791.

[21] Y. LeCun et al. "Comparison of Learning Algorithms for Handwritten Digit Recognition". In: *INTERNATIONAL CONFERENCE ON ARTIFICIAL NEURAL NETWORKS*. 1995, pp. 53–60.

[22] Li Wan et al. "Regularization of Neural Networks using DropConnect". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 2013, pp. 1058–1066. URL: http://proceedings.mlr.press/v28/wan13.html.

[23] François Chollet et al. *Keras*. https://keras.io. 2015.

[24] James Bergstra, Dan Yamins, and David D. Cox. *Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms*.

[25] https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html.

[26] Brian McFee et al. *librosa/librosa: 0.6.3*. Feb. 2019. DOI: 10.5281/zenodo.2564164. URL: https://doi.org/10.5281/zenodo.2564164.

[27] Brian McFee et al. *librosa/librosa: 0.6.3*. Feb. 2019. DOI: 10.5281/zenodo.2564164. URL: https://doi.org/10.5281/zenodo.2564164.

[28] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015.

[29] https://keras.io/applications.

[30] https://en.wikipedia.org/wiki/Beta_distribution.

TRITA -EECS-EX