Machine Learning manuscript No.
(will be inserted by the editor)

Symbolic DNN-Tuner

Michele Fraccaroli · Evelina Lamma · Fabrizio Riguzzi

the date of receipt and acceptance should be inserted later

Abstract Hyper-Parameter Optimization (HPO) occupies a fundamental role in Deep Learning systems due to the number of hyper-parameters (HPs) to be set. The state-of-the-art of HPO methods are Grid Search, Random Search and Bayesian Optimization. The first two methods try all possible combinations and random combination of the HPs values, respectively. This is performed in a blind manner, without any information for choosing the new set of HPs values. Bayesian Optimization (BO), instead, keeps track of past results and uses them to build a probabilistic model mapping HPs into a probability density of the objective function. Bayesian Optimization builds a surrogate probabilistic model of the objective function, finds the HPs values that perform best on the surrogate model and updates it with new results. In this paper, we improve BO applied to Deep Neural Network (DNN) by adding an analysis of the results of the network on training and validation sets. This analysis is performed by exploiting rule-based programming, and in particular by using Probabilistic Logic Programming. The resulting system, called Symbolic DNN-Tuner, logically evaluates the results obtained from the training and the validation phase and, by applying symbolic tuning rules, fixes the network architecture, and its HPs, therefore improving performance. We also show the effectiveness of the proposed approach, by an experimental evaluation on literature and real-life datasets.

M. Fraccaroli · E. Lamma
DE - Department of Engineering
University of Ferrara
Via Saragat 1, 44122 Ferrara, Italy
E-mail: {michele.fraccaroli, evelina.lamma}@unife.it

F. Riguzzi
DMI - Department of Mathematics and Computer Science
University of Ferrara
Via Saragat 1, 44122 Ferrara, Italy
E-mail: fabrizio.riguzzi@unife.it

Keywords Deep Learning · Hyper-Parameter Optimization · Probabilistic Logic Programming

1 Introduction

Deep Neural Networks (DNNs) are very sensitive to the tuning of their hyper-parameters (HPs). Different tunings of the same neural network can lead to completely different results. For this reason, Hyper-Parameter Optimization (HPOs) algorithms play an important role in building Deep Learning models. These algorithms have shown good performance [1], comparable with human experts.

This work aims at creating an algorithm to drive the training of DNNs, automatizing the choice of HPs and analysing the performance of each training experiment to obtain a network with better performance. The algorithm combines an automatic tuning approach with some tricks usually used in manual approaches [2]. For the automatic approach we use Bayesian Optimization (BO) [3]. This choice is motivated by the fact that tuning DNN is computationally expensive and the BO algorithm limits the evaluations of the objective function (DNNs training and validation in this case) by spending more time in choosing the next set of HPs values.

The whole software was written in Python, using TensorFlow for each part regarding the neural networks, Scikit-Optimize for implementing BO and ProbLog for the symbolic part of the software. The implementation is described in detail in [4].

The tricks used in manual approaches to solve problems are mapped into (non-deterministic, and probabilistic) Symbolic Tuning Rules (STRs). These rules identify Tuning Actions (TAs), which have the purpose of editing the HPs search space, adding new HPs or updating the network structure without human intervention. All this is aimed at avoiding network problems like overfitting, underfitting or incorrect learning rate values and driving the whole learning process to better results.

Symbolic DNN-Tuner is composed by two main parts: a Neural Block that manages the neural network, the HPs search space and the application of the TAs, and a Symbolic Block (developed with Probabilistic Logic Programming PLP, and STRs in particular) that, on the basis of the network performance and computed metrics after each training, diagnoses problems and identifies the (most probable) TA to be applied on the network architecture. In the beginning, probabilistic weights of STRs are set manually, and then they are refined, after each training, via Learning from Interpretations (an inference available in PLP) based on the improvements obtained or not, for each TA applied in previous training.

Therefore, our approach is positioned in the domain of Neural Symbolic systems, since it automatically tunes the DNN architecture by exploiting PLP and Learning from Interpretation, a kind of Inductive Logic Programming.

After a short discussion of preliminaries for a good understanding of the paper (Section 2), we introduce in Section 3 the main DNNs problems, the analysis done and the tuning rules used to improve the network architecture and performance. In Section 4 we present Symbolic DNN-Tuner with its building blocks and its execution pipeline. In Section 4.3 we present the symbolic section of Symbolic DNN-Tuner. More precisely, we show how, in the symbolic execution pipeline, Learning From Interpretation (LFI) is applied to learn the probability of the tuning rules and therefore dynamically change the probabilistic logic program. Experimental results are described in Section 5, in order to show that our approach greatly improves the network performance with respect to BO, either for literature and real-case datasets. Related work are discussed in Section 6.

2 Preliminaries

In this section, we review the preliminaries concepts, algorithm and approaches for a good understanding of the paper. We will deal with Bayesian Optimization (BO) like Hyper-parameters Optimization (HPO) algorithm (we will focus mainly on BO because is the HPO algorithm used in Symbolic DNN-Tuner), Probabilistic Logic Programming and Parameter Learning.

2.1 Bayesian Optimization

Bayesian Optimization (BO) is an approach to optimize objective functions (f) which are very expensive and/or slow to optimize [3]. The main idea behind this approach is to limit the time spent in the evaluation of f by spending more time choosing the new set of HPs values. BO builds a surrogate model of the objective function, quantifies the uncertainty in the surrogate using a regression model (*e.g., Gaussian Process Regression*) and uses an acquisition function to decide where to sample the new set of HPs [5]. The focus of BO is solving the problem:

$$\max_{x \in D} f(x) \tag{1}$$

where the input x is in $\mathbb{R}^d$, d is the number of HPs and D is a search space which can be seen as a hyper-cube where each dimension is a hyper-parameter. Then, BO builds a probabilistic model for $f(x)$ and exploits this model to decide where to sample the next set of HPs values. The idea is to use all the information derived from previous evaluations of $f(x)$ as a memory to make the next decision.

BO consists of two crucial components: the *probabilistic regression model* (e.g., Gaussian Process, see below) and the *activation function*. The first component provides a posterior probability distribution that captures the uncertainty in the surrogate model and the second determines the next point to evaluate. This is done by measuring the value that would be generated by the

$f(x)$ at this new point based on the posterior distribution [5]. This *activation function* is also used for finding a good balance between Exploration and Exploitation. Exploration aims at selecting samples that eliminate the parts of the input search space that do not include the maximizer of the $f(x)$, while Exploitation aims at selecting the sample closest to the optimum with a high probability [6].

Gaussian Processes (GPs) are stochastic processes and prior distributions on functions. GPs offers a non-parametric approach in that it finds a distribution over the possible functions $f(x)$ that are consistent with the observed data. A GPs can be used for regression problems. Any finite set $X = \{x_1, x_2, ..., x_n\}$ induces a multivariate Gaussian distribution with n dimensions and a GP is completely determined by the mean function μ and the covariance matrix K, $f \sim \mathcal{GP}(\mu, K)$ [7,8]. The covariance matrix K is created by evaluating a *covariance function* (*kernel*) k at each pair of points x_i, x_j, $K = k(x_i, x_j)$. The kernel is chosen so that points x_i and x_j that are closer in the input space, have a larger correlation. In this way, it can be obtained that these points should have more similar function values than points that are far apart. For convenience, we assume that the prior mean is the zero function $\mu = 0$ and, for covariance, a very popular choice is the squared exponential function

$$k(x_i, x_j) = \sigma^2 exp\Big(- \frac{1}{2l^2} \|x_i - x_j\|^2 \Big) \tag{2}$$

where parameters σ determines the variation of function values from their mean and l describes how smooth a function is. Given sets $X = \{x_1, x_2, ..., x_n\}^T$ and $Y = \{y_1, y_2, ..., y_n\}^T$ of observed values, the aim is to predict the y value for a new point x. It can be shown that y is Gaussian distributed with mean $\mu = \mathbf{k}^T C^{-1} Y$ and variance $\sigma^2 = k(x, x) - \mathbf{k}^T C^{-1} \mathbf{k}$ [9,10]. $\mathbf{k}$ is the column vector with elements $k(x_i, x)$ and C is composed by elements $C_{ij} = k(x_i, x_j) + s^2 \delta_{ij}$ where s^2 is the variance of the random noise in the linear regression model $y = f(x) + \epsilon$ and δ_{ij} is the *Kronecker delta* ($\delta_{ij} = 1$ if $i \neq j$, 0 if $i = j$). So, if $s^2 = 0$, $C = K$ (see Equation 2). In this way, it is possible to define a prior distribution over parameters instead of choosing values. For a complete and more precise overview of the Gaussian Process and its application to Machine Learning, see [7].

The Acquisition function determines the next point to evaluate or, in our case, the next set of HPs values. There are many types of acquisition function but, the most commonly used is *Expected Improvement* (EI) [5,11]. EI defines a non-negative expected improvement over the best previously observed target value at a given point x. Since we observe f, we can say that $f_n^* = \max_{m \leq n} f(x_m)$ is the optimal choice, i.e., the previously evaluated point with the largest value, where n is the number of times that we have evaluated f thus far. Now suppose that if we evaluate our function f at point x, we will observe $f(x)$. After the evaluation of at the point x ($f(x)$), the value of the best observed point will be either $f(x)$ or f_n^* (clearly, it depends on who is

greater than the other). The improvement in the value of the best observed point is:

$$best = \max\{0, f(x) - f_n^*\} \tag{3}$$

for simplicity we can write this like $[best]^+$. We would like to choose an x that leads to maximum improvement, and we take the expected value of this improvement and choose x to maximize it. We can formalize all of them like as follows:

$$EI_n(x) = E_n\left[[best]^+\right] \tag{4}$$

$$EI_n(x) = E_n\left[[f(x) - f_n^*]^+\right] \tag{5}$$

where $E_n[\cdot]$ is the expected value taken under the posterior distribution of f after having observed $x_1, ..., x_n$ [5].

2.2 Probabilistic Logic Programming

Probabilistic Logic Programming (PLP) is a tool for reasoning on uncertain relational domains that is gaining popularity in Statistical Relational Artificial Intelligence (StarAI) due to its expressiveness and intuitiveness. PLP has been successfully applied to a variety of fields, such as natural language processing [12,13,14], bio-informatics [15,16,17], link prediction in social networks [18], entity resolution [19] and model checking [20].

We consider ProbLog [16] for the simplicity of its syntax and the availability of a Learning From Interpretation module (Section 2.3). Moreover, the fact that it is written in simplifies the integration within Symbolic DNN-Tuner, that is written in Python as well. ProbLog is a PLP language under the distribution semantic [21]. A program that adopts a semantics of this type defines a probability distribution over normal logic programs called *worlds*. To define the probability of a query, this distribution is extended to a joint distribution of the query and the *worlds* and the probability of the query is obtained from the joint distribution by marginalization [10].

A ProbLog program consists of a set of clauses. Every clause c_i is labelled with the probability p_i. Listing 1 shows an example of ProbLog code.

Listing 1: ProbLog example

```
% Probabilistic facts:
0.5::heads1.
0.6::heads2.
% Rules:
twoHeads :- heads1, heads2.
query(heads1).
query(heads2).
query(twoHeads).
```

A ProbLog program $P = \{p_1 :: c_1, p_2 :: c_2, .., p_i :: c_m\}$ defines a probability distribution over logic program $L = \{c_1, c_2, ..., c_m\}$ and the aim of inference in ProbLog is to calculate the probability that the query will succeed [16]. With the reference at Listing 1, the three queries return the probability of 0.5, 0.6 and 0.3 for `heads`, `heads2` and `twoHeads` respectively.

ProbLog inference [22] works by generating all ground instances of clauses in the program the query depends on and transforming the clauses to a propositional formula. After that, the logic formula is compiled into a Sentential Decision Diagram (SDD) [23]. SDDs are a representation language for propositional knowledge bases. Then ProbLog evaluates the SDD bottom-up to calculate the success probability of the given query.

2.3 Parameter Learning

The ProbLog system [24] includes a Parameter Learning algorithm (LFI-ProbLog algorithm) [25] that learns the parameters of ProbLog programs from partial interpretations. Generally, one is interested in the maximum likelihood parameters given the training data. This can be formalized as follows:

Definition 1 (LFI-ProbLog learning problem) Given a ProbLog program P containing probabilistic facts with unknown parameters (probabilistic weights) and a set $E = \{\mathcal{I}_1, \ldots, \mathcal{I}_T\}$ of interpretations (the training examples), find the value of the parameters $\boldsymbol{\Pi}$ of P that maximize the likelihood of the examples, i.e., solve

$$\arg\max_{\boldsymbol{\Pi}} P(E) = \arg\max_{\boldsymbol{\Pi}} \prod_{t=1}^{T} P(q(\mathcal{I}_t))$$

where partial interpretation $\mathcal{I}$ can be written like $\mathcal{I} = \langle I_t, I_f \rangle$ where atoms in I_t are true and those in I_f are false. $\mathcal{I}$ can be associated with conjunction $q(\mathcal{I}) = \wedge_{a \in I_t} a \vee \wedge_{a \in I_f} \sim a$. So, given a ProbLog program and a series of partial interpretations $\mathcal{I}$ and evidence $q(\mathcal{I})$, the goal is to find the maximum likelihood parameters. Now, we need to pay attention to interpretations (the training examples) when computing $\arg\max_{\boldsymbol{\Pi}} P(E)$. If we have complete interpretations, the parameters (probabilistic weights) can be computed by relative frequency. If some interpretations in E are partial, instead, an EM algorithm [26] must be used [25,10]. For a more complete and precise reading on PLP and PLP Parameter Learning, see [10]. For a more detailed description of the application of LFI in Symbolic DNN-Tuner, see Listing 3 in Section 4.3.

3 DNNs Training Problems and Countermeasures

Experts in DNN training have identified several problems that can be encountered, see [2,27]. Here we list them together with the appropriate countermeasures mapped into Tuning Actions (TAs). Table 1 shows the association

Table 1: Symptoms and related problems

Symptoms	Problem
Gap between accuracy in training and validation	Overfitting
Gap between loss in training and validation	Overfitting
High loss	Underfitting
Low accuracy	Underfitting
Loss trend analysis	Increasing loss
Fluctuation of the loss	Fluctuating loss
Evaluation of the shape of the loss	Low learning rate
	High learning rate

Table 2: Problem - TA associations

Problem	Tuning Actions (TAs)	Acronyms
Overfitting	Regularization and Batch Normalization	reg_l2 & batch_norm
	Increase dropout	inc_dropout
	Data augmentation	data_augm
Underfitting	Decrease the learning rate	decr_lr
	Increase the number of neurons	inc_neurons
	Addition of fully connected layers	new_fc_layer
	Addition of convolutional blocks	new_conv_layer
Increasing loss	Decrease the learning rate	decr_lr_inc_loss
Fluctuating loss	Increase the batch size	inc_batch_size
	Decrease the learning rate	decr_lr_fl
Low learning rate	Increase learning rate	inc_lr
High learning rate	Decrease learning rate	dec_lr

between symptoms and diagnosed problems. Table 2 shows the association between the problems and the corresponding TAs with the acronyms used in Section 5 for the description of the experiments.

3.1 Overfitting

Overfitting is the lack of generalization ability of the model. This happens when the model adapts too much to the training data, not generalizing and therefore not working correctly on the validation data [27]. Diagnosing overfitting is relatively easy by monitoring the performance of the network during both training and validation. Symbolic DNN-Tuner checks the difference between training and validation phase for both the accuracy and loss, in order to identify any possible gap. A significant gap between the two phases is a clear symptom of overfitting.

When overfitting is diagnosed, Symbolic DNN-Tuner applies two possible TAs, as shown in Table 2: Regularization [28] and Batch Normalization [29], Increase Dropout [30] and Data Augmentation [31].

3.2 Underfitting

Underfitting happens when the model is not able to learn and fails in both training and validation phases [27]. In order to detect underfitting, Symbolic DNN-Tuner measures the accuracy and the loss in the validation phases. If, at a certain iteration, the loss is greater than a manually predetermined threshold and accuracy is lower than another manually predetermined threshold, Symbolic DNN-Tuner diagnoses underfitting. These two thresholds are dynamically increased as the algorithm progresses. This allows Symbolic DNN-Tuner to become increasingly demanding as iterations progress.

For fixing underfitting, there are different TAs (Table 2). Apart from decreasing the learning rate, the remaining TAs aim at increasing the learning capability of the network. Besides increasing the number of neurons and the addition of fully connected layers which are self explicative, we may add a convolutional block composed of a sequence of two convolutional layers followed by a pooling and dropout layer.

3.3 Increasing Loss

Symbolic DNN-Tuner uses Early Stopping, so, if at a certain iteration the loss starts growing, this means that the learning rate is probability too high. In this case, Symbolic DNN-Tuner applies a TA that aims at reducing the learning rate's search space by eliminating all values that are larger than the last chosen.

3.4 Fluctuating Loss

The oscillation in the loss is usually due to the batch size. When the batch size is 1, oscillation can occur and the loss will be noisy. When the batch size is the complete dataset, the oscillation will be minimal because each update of the gradient should improve the loss function monotonically (unless the learning rate is set too high).

When this behaviour is detected, a TA that aims at shrinking the batch size's search space is applied, to make sure larger values are selected in the next iterations.

3.5 Management of the Learning Rate

For the correct management of the learning rate, we exploit the relation between the loss and the learning rate [32], as can be seen in Figure 1. From the trend of the loss, Symbolic DNN-Tuner can diagnose if the learning rate is too high or too low. For doing this, the algorithm computes the integral of the loss and the line between the initial and the final loss, that is AUL and

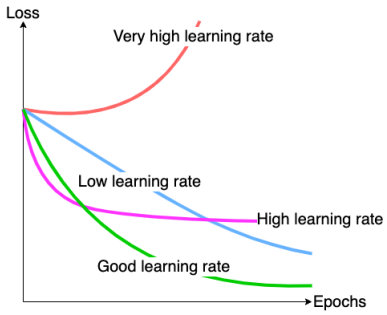


Fig. 1: Relation between loss and learning rate.

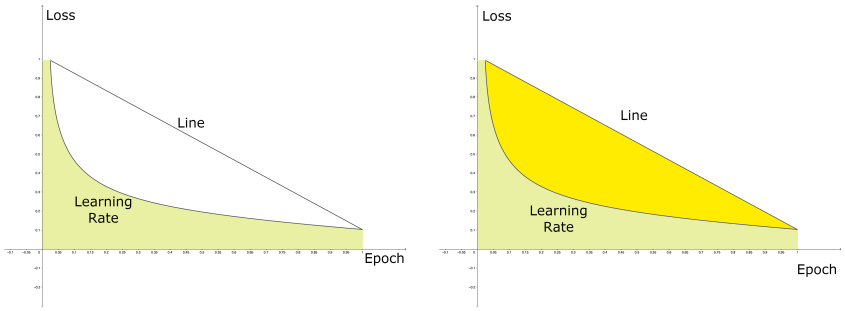
$AULL$ respectively. Then the absolute difference between the two is computed. The next step is to check whether the difference is greater or less than two thresholds as you can see in Equation 8.

$$AUL = \int loss \qquad AULL = \int line \tag{6}$$

$$R = |AULL - AUL| \tag{7}$$

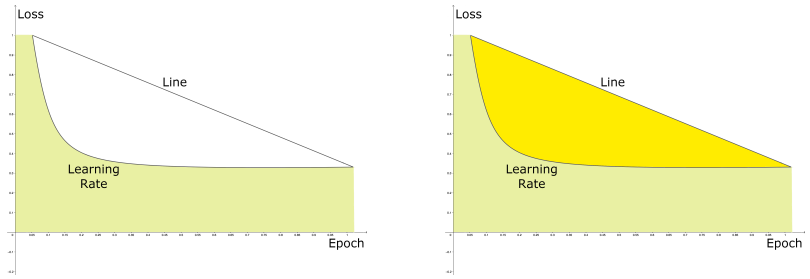
$$Problems = \begin{cases} too\_large\_lr & \text{if } R > \frac{3AULL}{4} \\ too\_small\_lr & \text{if } R < \frac{AULL}{4} \\ good\_lr & \text{otherwise} \end{cases} \tag{8}$$

Figures 3 and 4 shows the difference between the AUL and AULL in the three main cases of *good*, *high* and *low* learning rate. R changes considerably depending on the shape of the loss and therefore depending on the learning rate. When the diagnosis is too_large_lr, a TA that removes large values from the learning rates search space is applied. On the contrary, when the diagnosis is too_small_lr, a TA that removes the small values from the learning rates search space is applied.



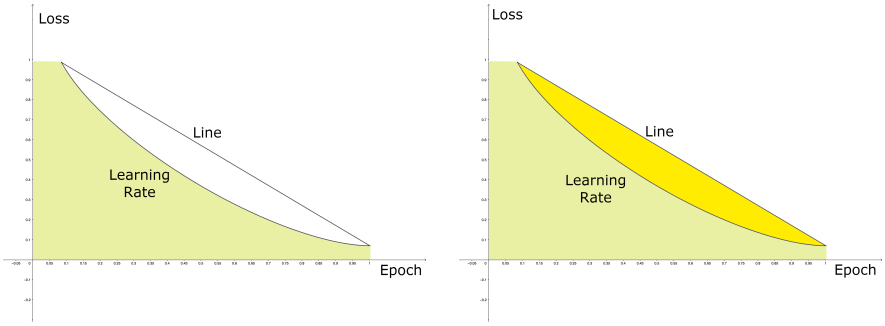
(a) AUL (b) AUL

Fig. 2: AUL (light yellow area) and difference displayed in Equation 7 (yellow area) when learning rate is *good*.



(a) AUL (b) $AUSL$

Fig. 3: AUL (light yellow area) and difference displayed in Equation 7 (yellow area) when learning rate is *high*.



(a) $AUSL$ (b) $AUSL$

Fig. 4: AUL (light yellow area) and difference displayed in Equation 7 (yellow area) when learning rate is *low*.

4 Symbolic DNN-Tuner

Symbolic DNN-Tuner is a system to drive the training of a Deep Neural Network, analysing the performance of each training experiment and automatizing the choice of HPs to obtain a network with better performance. It only requires an initial definition of the network architecture, a space of values for the HPs to be optimized and the dataset for training and validation. The system starts with a given set of rules with default weights (which change by LFI, after each training and diagnosis phase). These rules are written in PLP, and implement Table 2. A sample of these rules is given in Listing 2 and Listing 4.

Symbolic DNN-Tuner exploits BO for the choice of HPs and applies a performance analysis at the end of each training and validation session in order to identify possible problems such as overfitting, underfitting or incorrect learning rate configurations as described in Section 3.

By analyzing the behaviour of the network, it is possible to identify some problems (e.g., overfitting, underfitting, etc.) that BO is not able to avoid because it works only with a single metrics (validation loss or accuracy, training loss or accuracy). When Symbolic DNN-Tuner diagnoses these problems, it changes the search space of HP values or the architecture of the network by appling TAs to drive the DNN to a better solution.

4.1 Architecture

Symbolic DNN-Tuner is composed by two main parts: a *Neural Block* that manages the neural network, the HPs search space and the application of the TAs, and a *Symbolic Block*(where STRs are implemented in Probabilistic Logic Programming, PLP for short) that, on the basis of the network performance and computed metrics after each training, diagnoses problems and identifies the (most probable) TAs to be applied on the network architecture.

In the beginning, probabilistic weights of STRs are set manually, and then they are refined, after each training, via Learning from Interpretations (LFI) on the basis of the improvements obtained or not, for each TA applied in previous training. The schema of Symbolic DNN-Tuner and of its blocks is shown in Figure 5.

STRs are probabilistic rules that map problems into resolutive actions (TAs) as defined in Table 2, and their weights are learned by LFI [25].

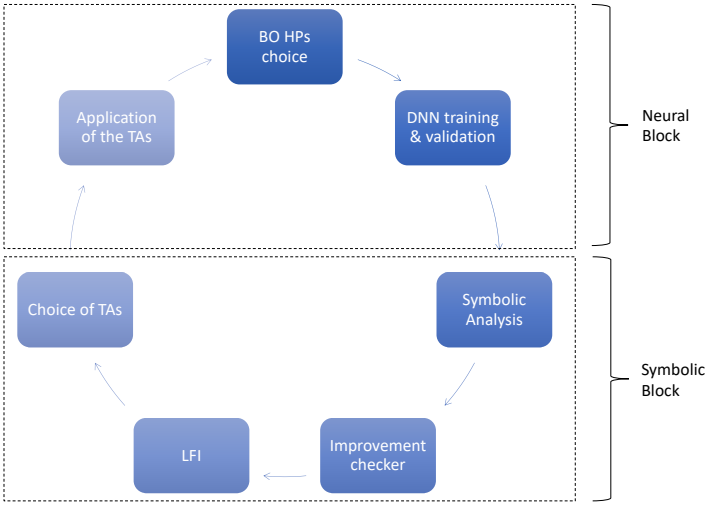


Fig. 5: Symbolic DNN-Tuner execution pipeline with *Neural Block* and *Symbolic Block*. In the Figure is shown the Symbolic DNN-Tuner's pipeline and the steps between Neural and Symbolic block.

4.2 Algorithm

At each iteration, Symbolic DNN-Tuner trains and validates the neural network means of BO. Once training and validation have been done, the algorithm checks if the training and validation have achieved better results than the previous training (the training of the neural network performed in the previous iteration of Symbolic DNN-Tuner) in terms of accuracy and loss (line 12 in Algorithm 1). This improvement check is used to build the training set for LFI. In fact, at each iteration, a new point is added to the training set for LFI and parameter learning is rerun. Then, the new weights of the STRs are placed in the symbolic program and the diagnosis starts. The symbolic analysis (line 22 in Algorithm 1) returns the TAs to be applied to the network architecture and/or HPs search space.

After each training, the BO status is saved, so that we can resume it for the new training of the network in the next iteration of Symbolic DNN-Tuner. The importance of starting a new training with a resumed BO status is that BO works by maintaining some kind of memory of the experience of past training and, in this way, it will choose the HPs values in an optimized way. This is possible only if the HPs search space or the neural network architecture has not changed.

Algorithm 1 Symbolic DNN-Tuner

procedure SYMBOLIC_DNN_TUNER(S, M, n, PM)
  $Iteration \leftarrow 0$
  $Ckpt \leftarrow \emptyset$
  $(Ckpt, R, H) \leftarrow BO_s(S, M)$
  $(NewM, NewS) \leftarrow$ANALYSIS_TUNING(R, H, PM)
  while $Iteration \leq n$ **do** ▷ Symbolic DNN-Tuner main loop
    if $NewM \neq M$ **then**
      $(Ckpt, R, H) \leftarrow BO_s(S, NewM)$ ▷ New training with BO from scratch
    else
      $(Ckpt, R, H) \leftarrow BO_r(NewS, NewM, Ckpt)$ ▷ New training with a restored
    end if ▷ - status of BO
    $Improve \leftarrow ImprovementChecker(R, DB)$
    $PM \leftarrow LearnFromInt(Improve, SymbolicDiagnosis, SymbolicTuning)$
    $(NewM, NewS) \leftarrow$ANALYSIS_TUNING(R, H, PM)
    $Iteration \leftarrow (Iteration + 1)$
  end while
end procedure

function ANALYSIS_TUNING(R, H, PM)
  $DB \leftarrow saveResult(DB, R)$
  $(AUL, AULL) \leftarrow Areas(H)$
  $(SymDiagnosis, SymTuning) \leftarrow SymbolicAnalysis([H, R, AUL, AULL], PM)$
  $(NewM, NewS) \leftarrow Tuning(SymDiagnosis, SymTuning)$
  return $(NewM, NewS)$
end function

Due to the sequential application of the TAs, there are some possible dangerous feedback loops. This problem can occur specifically with the TAs for the management of the learning rate (increasing and decreasing of the learning rate) or with the TAs to fix the underfitting. Thanks to the management described in Section 3.5, possible loops on learning rate are avoided. For TAs to fix the underfitting, thresholds have been set to prevent loops that lead to too large networks.

Algorithms 1 encapsulates the whole Symbolic DNN-Tuner's process. With BO_s and BO_r we refer respectively to the functions that applies BO from scratch and BO with resumed status respectively. With S, M, n and PM we refer respectively to the search space of HPs values, initial neural network, number of cycles of the algorithm and the *Probabilistic Model* coded into Symbolic DNN-Tuner. *Iteration* is the counter of the iteration of Symbolic DNN-Tuner. *Ckpt* is a checkpoint of the Bayesian Algorithm that can be used to restore the state of the BO.

BO_s, receiving S and M or $NewM$, returns the checkpoint of the BO, the results of the evaluation of the trained network in terms of loss and accuracy R (R is the tuple $(Acc, Loss)$) and the history H of the loss and accuracy in both training and validation. BO_r, receiving the new restricted search space $NewS$, the new neural network model $NewM$ and $Ckpt$, perform the same computation as BO_s but starting from a checkpoint rather than from scratch.

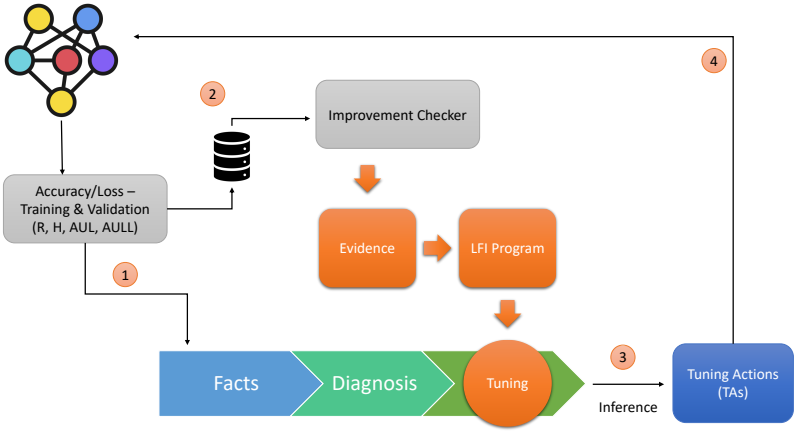


Fig. 6: Symbolic Block of Symbolic DNN-Tuner. The numbers mark the order of execution.

The results R are stored in the database DB. $Improve$ is a boolean value indicating whether there was an improvement over the previous iteration. PM is updated by the *Learning From Interpretation* ($LearnFromInt$) function after each iteration of the algorithm. This step can be applied after the first iteration of the whole algorithm because we need to have the $SymbolicDiagnosis$ and $SymbolicTuning$ to performing the $LearnFromInt$ function and learn the weights of the STRs. These two variables are obtained from the previous tuning step of the algorithm. The $LearnFromInt$ and $SymbolicAnalysis$ functions from line 19 of Algorithm 1 will be explained in detail in the next section. AUL and $AULL$ will be useful for the analysis of the learning rate. $NewM$ and $NewsS$ are the new model and the new restricted search space obtained after the application of the TA, respectively.

4.3 Symbolic Section

The Symbolic Block performs $LearnFromInt$ and $SymbolicAnalysis$ function in Algorithm 1. This block analyses the network metrics (R, H, AUL and $AULL$ in Algorithm 1), producing a diagnosis and, from this, returns the TAs to be applied. In the following, we describe in more detail how STRs have been implemented in PLP and how their weights are calibrated by exploiting LFI.

The Symbolic Block, is composed of a PLP program with three parts: *Facts*, *Diagnosis* and *Tuning* (`FACTS`, `DIAGNOSIS` and `TUNING` sections in Listing 4). A sample program is shown in Listing 4. The whole logic program is dynamically created by the union of these three parts at each iteration of the Algorithm 1, as can be seen in Figure 6. *Facts* memorizes R, H, AUL and $AULL$ obtained from the Neural Block (arc 1 in Figure 6). The *Diagnosis*

section encapsulates the code for diagnosing the DNNs behaviour problems. Finally, the *Tuning* section is composed by the STRs. *Facts*, *Diagnosis* and *Tuning* form the symbolic program. Thanks to ProbLog inference, we can query this program and obtain the TAs (arc 3 in Figure 6). And finally, TAs are passed to the Neural Block and applied on the DNN structure or the HPs search space.

Each STR encapsulates a TA associated with a problem (see the associations in Table 2). TAs and problems of Table 2 are mapped into arguments of symbolic tunining rules, occurring in their head and body, respectively, as shown in Listing 2. Each STR has a weight which determines the probability of application of its TA, in case the associated problem is diagnosed. In the *Tuning* section, each STR is a rule such as those described in Listing 2.

Listing 2: STRs in the symbolic part of Symbolic DNN-Tuner

```
0.7::action(data_augment):- problem(overfitting).
0.3::action(decr_lr):- problem(underfitting).
0.8::action(inc_neurons):- problem(underfitting).
0.4::action(new_conv_layer):- problem(underfitting).
```

Listing 2 shows a subset of all STRs (see *Tuning* section in Listing 4 for a complete version of Listing 2). The `problem(...)` predicate is defined in the *Diagnosis* section of the Symbolic Block, see Listing 4.

The probabilistic weights are learned from the *experience* (evidences) gained from previous iterations. This *experience* becomes the set of *training examples* for the LFI program. Then, the LFI program is composed of two parts: the program and the evidences obtained from the *ImprovementChecker* module, as shown in Listing 3:

Listing 3: Learning From Interpretation part of Symbolic DNN-Tuner

```
% Program
t(0.5)::action(data_augment).
t(0.2)::action(decr_lr).
t(0.85)::action(inc_neurons).
t(0.3)::action(new_conv_layer).
- - - - - - - - - - - - - - - - - - - - - -
% Evidence
evidence(action(data_augment), True).
evidence(action(decr_lr), False).
```

This file is built dynamically at each iteration of Algorithm 1. After each training, Symbolic DNN-Tuner checks the improvement of the network with the *ImprovementChecker* module. The improvement (*Improve* in the Algorithm 1) is a Boolean value and it is used to build the evidence. The aim is to reward with greater probability those TAs that have led to improvements. In detail, in Figure 7 we can see that, starting from a program like Listing

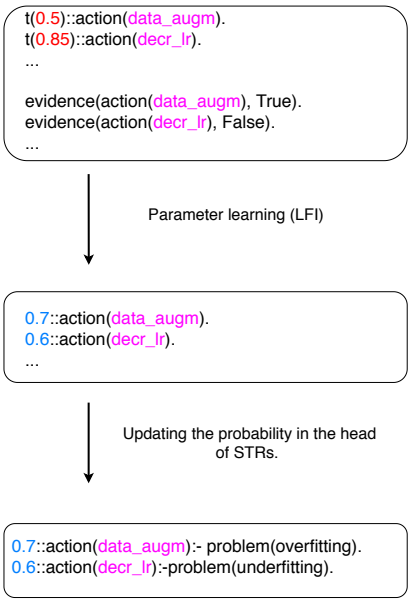


Fig. 7: Learning From Interpretation pipeline. In the middle rectangle, in blue, we can see the learned parameters after the LFI.

3, learning the parameters of this program, we can obtain the new values of probability of applying the TA. After that, we can update the probability in the head of the STRs. In this way, Symbolic DNN-Tuner can learn which TA was better and consequently favours it over the others.

Finally, with the complete and updated symbolic program, we can use ProbLog inference and query the program for the probabilities of given query atoms, say, `query(problem(_))` and `query(action(_))`. For clarity, an extract of ProbLog code is provided in Listing 4.

In the rest of this Section, we show an extract of ProbLog code used in Symbolic DNN-Tuner.

Listing 4: Extract of Logic section of Symbolic DNN-Tuner

```
% FACTS ----------------------------------------------------------
a([0.0529399998486042, 0.0710360012948513,
0.6266616476927525, 0.6298289950701192, ... ]).
va([0.0191, 0.0593, 0.1797, 0.2304, 0.2512, 0.28,
0.5261, 0.5339, 0.5273, ... ]).
l([4.776382889623642, 4.218988112640381, 3.960466429057121,
```

```
1.8257129939079284, ... ]).
vl([5.670237278938293, 4.4710672222614285,
2.000358765614033, 1.9812814263105392, ...]).
itacc(0.10625000000000001).
itloss(0.4125).

% DIAGNOSIS ------------------------------------------------------
abs2(X,Y) :- Y is abs(X).
isclose(X,Y,W) :- D is X - Y, abs2(D,D1), D1 =< W.
gap_tr_te_acc :- a(A), va(VA), last(A,LTA), last(VA,ScoreA),
Res is LTA - ScoreA, abs2(Res,Res1), Res1 > 0.2.
gap_tr_te_loss :- l(L), vl(VL), last(L,LTL), last(VL,ScoreL),
Res is LTL - ScoreL, abs2(Res,Res1), Res1 > 0.2.
low_acc :- va(A), itacc(Tha), last(A,LTA),
Res is LTA - 1.0, abs2(Res,Res1), Res1 > Tha
high_loss :- vl(L), itloss(Thl), last(L,LTL), \+isclose(LTL,0,Thl).

% PROBLEMS -------------------------------------------------------
problem(overfitting) :- gap_tr_te_acc; gap_tr_te_loss.
problem(underfitting) :- high_loss; low_acc.

% TUNING ---------------------------------------------------------
action(reg_l2) :- problem(overfitting).
0.545454545454545::action(inc_dropout):- problem(overfitting).
0.0::action(data_augment):- problem(overfitting).
0.3::action(decr_lr):- problem(underfitting).
0.0::action(inc_neurons):- problem(underfitting).
0.545454545454545::action(new_fc_layer):- problem(underfitting).
0.4::action(new_conv_layer):- problem(underfitting).

% QUERY ----------------------------------------------------------
query(problem(_)).
query(action(_)).
```

Listing 4 shows a portion of the logic program of Symbolic DNN-Tuner. The first part contains the *Facts*. They describe the history of the accuracy and the loss during training phase and validation phase (`a([])`, `l([])` and `va([])` and `vl([])` respectively). `itacc()` and `itloss()` are two threshold used to diagnose underfitting (see Section 3).

The *Diagnosis* section contains some utility functions and examples of clauses used for the analysis applied to *Facts*. The clauses are used to catch some gaps between training and validation phases of accuracy and loss, and to identify if loss is too high or accuracy too low. With these clauses, we can identify the problems encountered through the clauses found in the *Problems* section.

At the end, there are the STRs containing the tuning actions (TAs) (*Tuning* section), each with its probability in the head of the clauses. For example, if `gap_tr_te_acc` or `gap_tr_te_loss` is **true**, this means that there is a gap between training and validation accuracy or loss larger than 0.2 (e.g., training accuracy is 0.8 and validation accuracy is 0.5). This means that `problem(overfitting)` is **true**, then *overfitting* is diagnosed and the clauses with body true are the following in the Listing 5:

Listing 5: Clauses with body true

```
action(reg_l2) :- problem(overfitting).
0.545454545454545::action(inc_dropout):- problem(overfitting).
0.0::action(data_augment):- problem(overfitting).
```

By querying the program with `query(problem(_))` and `query(action(_))` we retrieve the problems and the TAs. In this case we retrieve overfitting with `query(problem(_))` and L2 regularization, increment dropout and data augmentation each with its probability of application with `query(actions(_))`.

5 Experiments

In this section we present the results obtained from the various experiments. The code is available at `https://github.com/micheleFraccaroli/Symbolic_DNN-Tuner.git`. Symbolic DNN-Tuner was tested on three different datasets: CIFAR10 [33], CIFAR100 [34] and CIMA_CIM, and compared to classic BO. On CIFAR10, Symbolic DNN-Tuner was also compared with *Efficient Neural Architecture Search* (ENAS) [35], Differentiable Architecture Search (DARTS) [36] and Autokeras[1] which is one of the most widely used AutoML systems [37]. The ENAS experiments exploits the *micro* search space and *macro* search space [35]. All NAS algorithms was implemented with Neural Network Intelligence (NNI)[2] toolkit provided by Microsoft, except Autokeras. In these experiments, Convolutional Neural Networks (CNNs) were used as DNNs.

CIFAR10 contains 60000 32x32 color images divided in 10 classes. In this experiment we have used CIFAR10 with 50000 training images and 10000 validation images. CIFAR100 is the same of CIFAR10 but divided in 100 classes instead of 10. CIMA_CIM is a dataset provided by CIMA S.P.A[3] with 3200 training images and 640 of validation images of size 256x128. These images are facsimiles of Euro-like banknotes. CIMA_CIM has 16 classes that represent the denomination and orientation of the banknote (e.g., 5_front, 5_rear, 10_front, 10_rear, etc).

All experiments were performed on the GALILEO cluster provided by Cineca [4], equipped with Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz and Nvidia K80 GPUs. For each experiment, early stopping was set. Each experiment had a fixed duration of 8 hours. This means that Symbolic DNN-Tuner, BO and the NASs algorithms run for 8 hours consecutively at most.

The first experiment was performed using the CIFAR10 dataset. Symbolic DNN-Tuner and BO start with a neural network with two blocks composed by two convolutional layers, a max-pooling layer, a dropout and two fully connected layers separated by a dropout layer, the second fully connected layer is the output. The initial hyper-parameters to be set by the algorithm are the number of the neurons in the convolutional and fully connected layers, the values of the Dropout layers, the learning rate, the batch size, the activation functions and the optimizator. The size of the search space depends

[1] Autokeras: https://autokeras.com/
[2] NNI: https://www.microsoft.com/en-us/research/project/neural-network-intelligence/
[3] CIMA: http://www.cima-cash-handling.com/it/
[4] Cineca: https://www.cineca.it/

Table 3: CIFAR10 - Symbolic DNN-Tuner

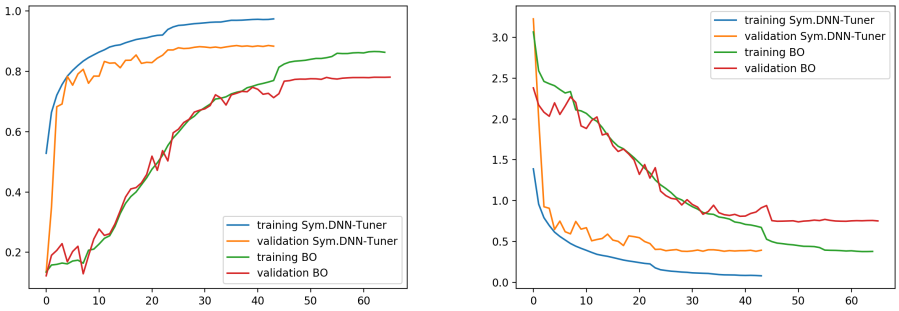
Step	Training		validation		Diagnosis	TA
	Accuracy	Loss	Accuracy	Loss		
1	0.9315	0.1958	0.793	0.7004	overfitting	reg_l2 & batch_norm data_augm
					underfitting	inc_neurons
2	0.943	0.1648	0.8812	0.3658	overfitting	reg_l2 & batch_norm data_augm
					floating_loss	inc_batch_size
3	0.8715	0.365	0.8364	0.4968	underfitting	inc_neurons
					floating_loss	decr_lr
4	0.9565	0.1287	0.8578	0.4742	overfitting	reg_l2 & batch_norm data_augm
					floating_loss	decr_lr
5	0.8967	0.2946	0.8759	0.3711	floating_loss	decr_lr
6	0.8902	0.3129	0.8655	0.3916	floating_loss	decr_lr
7	0.9435	0.1678	0.8692	0.3900	overfitting	reg_l2 & batch_norm data_augm
					underfitting	inc_neurons
					floating_loss	decr_lr
8	0.9549	0.1348	0.8768	0.4021	overfitting	reg_l2 & batch_norm data_augm
					floating_loss	decr_lr
9	0.9699	0.0999	0.8731	0.408	overfitting	reg_l2 & batch_norm data_augm
					underfitting	inc_neurons
					floating_loss	decr_lr
10	**0.9741**	**0.07971**	**0.8836**	**0.3925**	**overfitting**	**reg_l2 & batch_norm data_augm**
					underfitting	**inc_neurons**
					floating_loss	**decr_lr**

on hyper-parameter. The domains of HPs are as follows. The size of the first and second convolutional layers are between 16 and 64 and between 64 and 128 respectively. The domains of the rate of dropout for the first and second convolutional layers are [0.002,0.3] and [0.03,0.5] respectively. The size of the fully connected layers is between 256 and 512. The domain of the learning rate is $[10^{-5},10^{-1}]$. The choice for the activation functions are: *ReLU*, *ELU* and *SELU*. The choice for optimizers are: *Adam, Adamax, RMSprop* and *Adadelta*. For ENAS ([35] Sect. 3.2) and DARTS ([36] Appendix A.1.1), NNI uses the default search space in the original paper for CIFAR10. Autokeras starts with three-layers CNN. Each convolutional layer is actually a convolutional block of a ReLU layer, a batch-normalization layer, the convolutional layer, and a pooling layer. All the convolutional layers are with kernel size equal to three, stride equal to one, and number of filters equal to 64 [37].

In Table 3 (in bold the best result) we show the results of the execution of Symbolic DNN-Tuner for every iteration: accuracy and loss in both phases, the diagnosis of this iteration and the TAs for the next iteration (i.e., the results of step 2 is the result of the application of TA of step 1). As can be seen, the

Table 4: CIFAR10 - Bayesian Optimization

Step	Training		Validation	
	Accuracy	Loss	Accuracy	Loss
1	0.9762	0.06377	0.7776	1.046
2	0.09708	2.307	0.1	2.3078
3	0.1008	4.107	0.1	3.72
4	0.9133	0.2379	0.7655	0.9343
5	0.09874	2.303	0.1	2.303
6	0.7834	0.618	0.7404	0.7616
7	0.09842	2.303	0.1	2.303
8	0.1008	8.641	0.1	3.377
9	0.7599	0.6915	0.718	0.8171
10	**0.888**	**0.3272**	**0.7824**	**0.8229**
11	0.9609	0.1094	0.7758	1.036
12	0.6628	0.9813	0.6419	1.037
13	0.102	2.337	0.1	2.329
14	0.7411	0.7453	0.7327	0.7729
15	0.8633	0.378	0.7809	0.7518



(a) Accuracy (b) Loss

Fig. 8: Best results of Symbolic DNN-Tuner and Bayesian Optimization on CIFAR10.

f

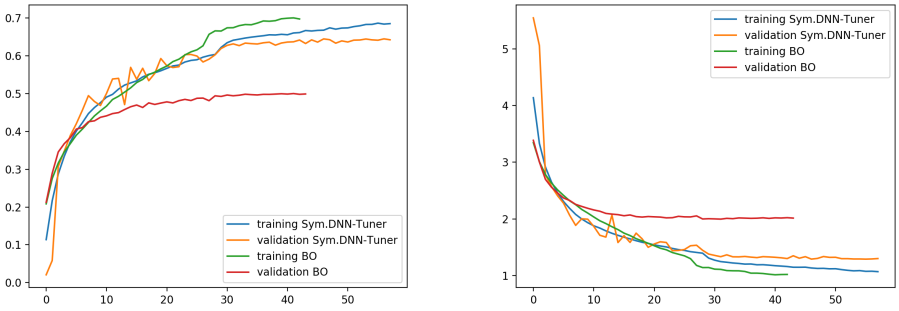
TA `reg_l2 & batch_norm` to fix the overfitting is always applied regardless of other TA once overfitting is diagnosed. This is because using a regularization and a batch normalization in any case does not lead to worsening.

In Table 4 (in bold the best result) we show the results of the application of standard BO on the same network and same dataset of the previous experiment. Figure 8 compares the best results of Symbolic DNN-Tuner and BO on CIFAR10 graphically.

Table 5 show a recap of the experiments performed on the dataset CIFAR10. It compares Symbolic DNN-Tuner with standard BO, ENAS with both Macro and Micro search space, DARTS and Autokeras. Only DARTS

Table 5: CIFAR10 - Comparison algorithms

Algorithms	Val. accuracy	Val. loss
Symbolic DNN-Tuner	0.8836	0.3925
Bayesian Opt.	0.7824	0.8229
ENAS Macro search	0.4520	1.6540
ENAS Micro search	0.4887	1.6711
DARTS	0.9554	0.1526
Autokeras	0.9458	0.2507



(a) Accuracy (b) Loss

Fig. 9: Best results of Symbolic DNN-Tuner and Bayesian Optimization on CIFAR100.

and Autokeras outperform Symbolic DNN-Tuner in 8 hours of execution on CIFAR10.

From the CIFAR10 experiment, we can see that Symbolic DNN-Tuner does not always obtain the networks with the best performances but, unlike the BO and NAS methods, Symbolic DNN-Tuner guarantees an explanation of the actions it performs during its operation as shown in Table 3. At each iteration, we can see that a certain TA is applied to address a certain problem that is identified during the diagnosis phase.

The second experiment was performed on the CIFAR100 dataset. The initial network is the same as in the previous experiment. Tables 6 and 7 show the progression of Symbolic DNN-Tuner and BO respectively (in bold the best results), and Figure 9 compares the best results of Symbolic DNN-Tuner and BO graphically. Table 6 shows that the increase in the number of classes w.r.t. the number of images in the dataset leads to a worsening of the quality of the training in terms of accuracy and loss. By comparing Table 6 and Table 7 it can be seen how Symbolic DNN-Tuner outperform BO in terms of both accuracy and loss on CIFAR100.

Table 6: CIFAR100 - Symbolic DNN-Tuner

Step	Training		validation		Diagnosis	STR
	Accuracy	Loss	Accuracy	Loss		
1	0.7121	1.039	0.4067	2.624	overfitting underfitting	reg_l2 & batch_norm data_augm inc_neurons
2	0.7199	0.98	0.6238	1.443	overfitting underfitting floating_loss	reg_l2 & batch_norm data_augm inc_neurons inc_batch_size
3	0.7953	0.6752	0.6496	1.33	underfitting underfitting floating_loss	inc_neurons data_augm inc_neurons inc_batch_size
4	0.8353	0.5786	0.6056	1.548	overfitting underfitting	reg_l2 & batch_norm data_augm inc_neurons
5	0.6656	1.157	0.6324	1.309	underfitting floating_loss	inc_neurons inc_batch_size
6	**0.6852**	**1.068**	**0.6425**	**1.299**	**overfitting** **underfitting** **floating_loss**	**reg_l2 & batch_norm** **data_augm** **inc_neurons** **inc_batch_size**
7	0.663	1.156	0.6362	1.324	underfitting floating_loss	inc_neurons inc_batch_size

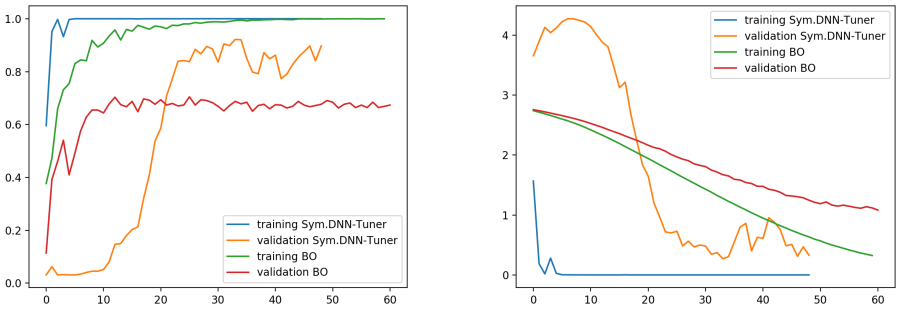
Table 7: CIFAR100 - Bayesian Optimization

Step	Training		Validation	
	Accuracy	Loss	Accuracy	Loss
1	$9.62e-3$	5.628	$1e-2$	5.691
2	0.475	2.017	0.4065	2.391
3	$9.62e-3$	4.935	$1e-2$	4.768
4	$9.86e-3$	5.087	$1e-2$	5.035
5	0.4634	2.061	0.4247	2.26
6	0.4981	1.911	0.4107	2.365
7	0.9612	0.13	0.4487	3.446
8	$9.82e-3$	4.698	$1e-2$	4.685
9	0.8432	0.5023	0.4594	2.524
10	0.7943	0.7538	0.4216	2.633
11	0.8226	0.5973	0.3923	2.956
12	0.8304	0.5655	0.3869	3.729
13	**0.692**	**1.056**	**0.4901**	**2.079**
14	0.8143	0.6264	0.4472	2.692
15	0.9771	0.08094	0.4483	3.766

The last two experiments was performed on the CIMA_CIM dataset. This was used to test Symbolic DNN-Tuner on an industrial, real case. To make this experiment as similar as possible to a production environment, the CIMA_CIM validation dataset has been degraded. In this experiment, in 8 hours, both Symbolic DNN-Tuner and BO have performed more than fifty trainings, then, for simplicity, only the best trainings from the experiment will be shown for both systems.

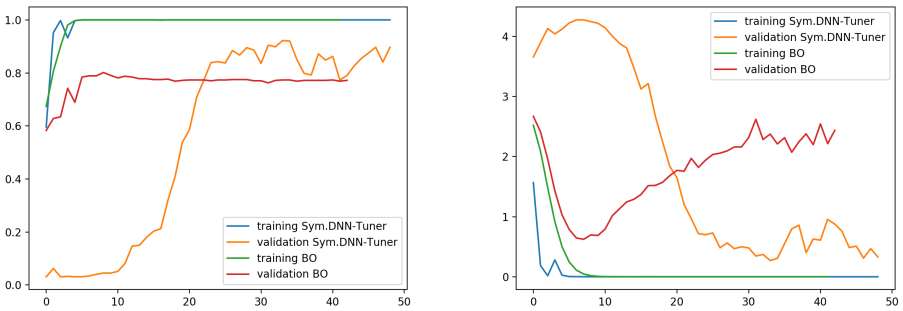
In the first experiment on the CIMA_CIM dataset, both Symbolic DNN-Tuner and BO starts with a small neural network with only one convolutional block and, at the end of the network, two fully connected layers divided by a dropout layer, the second fully connected layer being the output. The results are shown in Figure 10. In this experiment, Symbolic DNN-Tuner starts with the network previously described and ends with a network with two convolutional blocks (two convolutional layers, a max-pooling layer and dropout at the end) and at the end of the network, four fully connected layers plus a dropout layer and a fully connected layer for the output. BO, instead, keeps the same network because is not able to modify the architecture of the network. The network obtained with Symbolic DNN-Tuner shows a irregular behaviour in the first part of training that is more fluctuating than the one obtained with BO. After the 25th training cycle, we note how the network obtained by Symbolic DNN-Tuner shows a higher accuracy and a lower loss than that obtained by BO (Figure 10).

In the second experiment on the CIMA_CIM dataset, Symbolic DNN-Tuner produces the same network as the first experiment on this dataset. BO, instead, starts with a neural network with two blocks composed by two convolutional layers, a max-pooling layer, a dropout and two fully connected layers separated by a dropout layer, the second fully connected layer being the output. The results are shown in Figure 11. As in the previous experiment, Symbolic DNN-Tuner shows a behaviour in the first part of training that is more fluctuating than the one obtained with BO. The network obtained by BO shows the best results around the 10th training cycle. After that, it begins to deteriorate showing evident signs of overfitting (Figure 11).



(a) Accuracy (b) Loss

Fig. 10: Best results of Symbolic DNN-Tuner and Bayesian Optimization on CIMA_CIM dataset in the the part one of the experiment.



(a) Accuracy (b) Loss

Fig. 11: Best results of Symbolic DNN-Tuner and Bayesian Optimization on CIMA_CIM dataset in the part two of the experiment.

In all the performed experiments, it can be seen that Symbolic DNN-Tuner shows better results by analysing the performance of each network that it trains. See Figures 10 and 11.

6 Related Work

In the Machine Learning automation scenario, we can distinguish two main work areas: the HPO algorithm and the Neural Architecture Search (NAS) algorithm [38]. The first works only on the HPs that govern the main settings of

Machine Learning systems (including Deep Learning) for the training phases. The latter works mainly on the DNNs architecture.

In the field of Deep Learning, the state-of-the-art of HPO algorithm are: Grid Search, Random Search and BO. Grid search [39] is the basic method for the HPO. It performs exhaustive research (also called brute-force research) on the user-specified HPs search space. This algorithm performs new training for each combination of the HPs and each training is independent of the others. This allows it to run in parallel and guarantees to find the optimal configuration but, Grid Search suffers from the *curse of dimensionality*. This problem arises because the computational resources increase exponentially with the number of hyper-parameters to set [39]. The application of this algorithm with the actual DNNs is correlated with the huge amount of HPs to set (then a huge number of possible configuration) and to the dimension of the modern DNNs architecture which could take a long time to complete the training phase. This rise a time problem.

Random Search [40] performs a random search over the used defined HPs search space. Random search leads to better results than the previous algorithm due to the predetermined budget (the searching process stops when this budget is reached). Random search may perform better especially when some HPs are not uniformly distributed [39]. Unlike the Grid Search, this algorithm does not guarantee to achieve the optimum, but it requires less computational time while finding a reasonably good model in most cases [40].

BO is a Sequential Model-Based Optimization (SMBO) algorithm aimed at finding the global optimum with the minimum number of trials. Its success in optimizing the HPs of DNNs is because BO limits the number of training of DNNs spending more time choosing the next set of HPs to try. For a most detailed description of BO see Section 2. In literature, there are works that apply this kind of HPO algorithm to DNNs [41,8,42].

NAS is the process of automating the design of DNNs architectures. It is strictly correlated to HPO and AutoML. NAS methods have outperformed manually designed architectures [43,44]. In literature, there are different approaches to discover new neural architectures. Given a *search space* for a NAS, that is which neural architectures a NAS approach might discover, there are different *search strategies* can be used to explore the space of neural architectures. These strategies include: random search, BO [37], reinforcement learning [35], gradient-based methods [36] and evolutionary algorithms [45] [38]. The three main concepts of NAS are: *Search Space, Search Strategy* and *Performance Estimation Strategy*. *Search Space* refers to all possible architectures that can be generated by the NAS. *Search Strategy* refers to the methods to explore the search space with the canonical exploration-exploitation trade-off. *Performance Estimation Strategy* refers to the methods to measure the performance of the built neural network [38]. We can group the NAS in two families: the classical NAS and the one-shot NAS [46]. Classical NAS uses the traditional search space approach, where each generated DNN runs as an in-

dependent run. One-shot NAS algorithms use weight sharing among models in neural architecture search space to train a supernet. That supernet embraces many options for the final design rather than generating and training several DNNs independently and uses this to select better models. This type of algorithms reduces computational resource compared to the classical NAS algorithm and are differentiable, allowing the use of gradient descent to optimize them. The state-of-the-art of one-shot NAS are: Efficient Neural Architecture Search (ENAS) [35], Differentiable Architecture Search (DARTS) [36], Single Path One-Shot (SPOS) [47] and ProxylessNAS [48].

7 Conclusions

We have presented a system to automatically drive the training of a Deep Neural Network, by automatizing the choice of hyper-parameters in order to obtain a network with the best possible performance. This was achieved by combining Bayesian Optimization with an analysis of the network performance implemented by exploiting rule-based programming. In particular, tuning rules have been implemented in the ProbLog language, and their weights calibrated — after each training — by exploiting Learning from Interpretation. Symbolic DNN-Tuner thus exploits probabilistic symbolic rules that identify, after each training, the most appropriate tuning actions in response to diagnosed problems. These tuning actions restrict the hyper-parameters search space and/or update the network architecture without any human intervention. The experiments show that Symbolic DNN-Tuner performs better than standard Bayesian Optimization in terms of accuracy and loss, and also provides an *explanation* of the possible reasons for network malfunctioning.

Acknowledgements The authors want to thank CIMA S.P.A for providing a real-use case dataset to test the software developed in this work. The first author is supported by a PhD scholarship funded by Emilia-Romagna region, under POR FSE 2014-2020 program. Authors also acknowledge "SUPER: Supercomputing Unified Platform - Emilia-Romagna" project, financed under POR FESR 2014-2020. This work was partly supported by the "National Group of Computing Science (GNCSINDAM)".

References

1. Bergstra, J.S., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems, pp. 2546–2554 (2011)
2. Montavon, G., Orr, G., Múller, K.R.: Neural networks: tricks of the trade, vol. 7700. springer (2012)
3. Dewancker, I., McCourt, M., Clark, S.: Bayesian optimization primer (2015)
4. Michele Fraccaroli Evelina Lamma, F.R.: Symbolic dnn-tuner, a python and problog-based system for optimizing deep neural networks hyperparameters. SoftwareX (2021). Under submission
5. Frazier, P.I.: A tutorial on bayesian optimization. arXiv preprint arXiv:1807.02811 (2018)
6. Jalali, A., Azimi, J., Fern, X.Z.: Exploration vs exploitation in bayesian optimization. CoRR **abs/1204.0047** (2012). URL `http://arxiv.org/abs/1204.0047`

7. Rasmussen, C.E.: Gaussian processes in machine learning. In: Summer School on Machine Learning, pp. 63–71. Springer (2003)
8. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Advances in neural information processing systems, pp. 2951–2959 (2012)
9. Bishop, C.M.: Pattern recognition and machine learning. springer (2006)
10. Riguzzi, F.: Foundations of Probabilistic Logic Programming. River Publishers (2018)
11. Jones, D.R., Schonlau, M., Welch, W.J.: Efficient global optimization of expensive black-box functions. Journal of Global optimization **13**(4), 455–492 (1998)
12. Sato, T., Kubota, K.: Viterbi training in prism. Theory and Practice of Logic Programming **15**(2), 147–168 (2015)
13. Riguzzi, F., Lamma, E., Alberti, M., Bellodi, E., Zese, R., Cota, G., et al.: Probabilistic logic programming for natural language processing. In: URANIA@ AI* IA, pp. 30–37 (2016)
14. Fadja, A.N., Riguzzi, F.: Probabilistic logic programming in action. In: Towards integrative machine learning and knowledge extraction, pp. 89–116. Springer (2017)
15. Mørk, S., Holmes, I.: Evaluating bacterial gene-finding hmm structures as probabilistic logic programs. Bioinformatics **28**(5), 636–642 (2012)
16. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: IJCAI, vol. 7, pp. 2462–2467. Hyderabad (2007)
17. Sato, T., Kameya, Y.: Prism: a language for symbolic-statistical modeling. In: IJCAI, vol. 97, pp. 1330–1339 (1997)
18. Meert, W., Struyf, J., Blockeel, H.: Cp-logic theory inference with contextual variable elimination and comparison to bdd based inference methods. In: International Conference on Inductive Logic Programming, pp. 96–109. Springer (2009)
19. Riguzzi, F.: Speeding up inference for probabilistic logic programs. The Computer Journal **57**(3), 347–363 (2014)
20. Gorlin, A., Ramakrishnan, C., Smolka, S.A.: Model checking with probabilistic tabled logic programming. Theory and Practice of Logic Programming **12**(4-5), 681–700 (2012)
21. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: In Proceedings of the 12th International Conference On Logic Programming (ICLP'95. Citeseer (1995)
22. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory and Practice of Logic Programming **15**(3), 358–401 (2015)
23. Darwiche, A.: Sdd: A new canonical representation of propositional knowledge bases. In: Twenty-Second International Joint Conference on Artificial Intelligence (2011)
24. Dries, A., Kimmig, A., Meert, W., Renkens, J., Van den Broeck, G., Vlasselaer, J., De Raedt, L.: Problog2: Probabilistic logic programming. In: Joint european conference on machine learning and knowledge discovery in databases, pp. 312–315. Springer (2015)
25. Gutmann, B., Thon, I., De Raedt, L.: Learning the parameters of probabilistic logic programs from interpretations. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 581–596. Springer (2011)
26. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the em algorithm. Journal of the Royal Statistical Society: Series B (Methodological) **39**(1), 1–22 (1977)
27. Andrew Ngn Younes Bensouda Mourri, K.K.: Improving deep neural networks: Hyperparameter tuning, regularization and optimization
28. van Laarhoven, T.: L2 regularization versus batch and weight normalization. CoRR **abs/1706.05350** (2017). URL `http://arxiv.org/abs/1706.05350`
29. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. CoRR **abs/1502.03167** (2015). URL `http://arxiv.org/abs/1502.03167`
30. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research **15**(1), 1929–1958 (2014)
31. Shorten, C., Khoshgoftaar, T.M.: A survey on image data augmentation for deep learning. Journal of Big Data **6**(1), 60 (2019)

32. Ou, M., Wei, H., Zhang, Y., Tan, J.: A dynamic adam based deep neural network for fault diagnosis of oil-immersed power transformers. Energies **12**(6), 995 (2019)
33. Krizhevsky, A., Nair, V., Hinton, G.: Cifar-10 (canadian institute for advanced research) URL `http://www.cs.toronto.edu/~kriz/cifar.html`
34. Krizhevsky, A., Nair, V., Hinton, G.: Cifar-100 (canadian institute for advanced research) URL `http://www.cs.toronto.edu/~kriz/cifar.html`
35. Pham, H., Guan, M.Y., Zoph, B., Le, Q.V., Dean, J.: Efficient neural architecture search via parameter sharing. arXiv preprint arXiv:1802.03268 (2018)
36. Liu, H., Simonyan, K., Yang, Y.: Darts: Differentiable architecture search. arXiv preprint arXiv:1806.09055 (2018)
37. Jin, H., Song, Q., Hu, X.: Auto-keras: An efficient neural architecture search system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 1946–1956 (2019)
38. Elsken, T., Metzen, J.H., Hutter, F.: Neural architecture search: A survey. arXiv preprint arXiv:1808.05377 (2018)
39. Yu, T., Zhu, H.: Hyper-parameter optimization: A review of algorithms and applications. arXiv preprint arXiv:2003.05689 (2020)
40. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of Machine Learning Research **13**(Feb), 281–305 (2012)
41. Korichi, Guillemot, M., Heusèle, C., Rodolphe: Tuning neural network hyperparameters through bayesian optimization and application to cosmetic formulation data. In: ORASIS 2019 (2019)
42. Bertrand, H., Ardon, R., Perrot, M., Bloch, I.: Hyperparameter optimization of deep neural networks: Combining hyperband with bayesian model selection. In: Conférence sur l'Apprentissage Automatique (2017)
43. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Aging evolution for image classifier architecture search. In: AAAI 2019 (2019)
44. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 8697–8710 (2018)
45. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. In: Proceedings of the aaai conference on artificial intelligence, vol. 33, pp. 4780–4789 (2019)
46. Bender, G., Kindermans, P.J., Zoph, B., Vasudevan, V., Le, Q.: Understanding and simplifying one-shot architecture search. In: International Conference on Machine Learning, pp. 550–559 (2018)
47. Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., Sun, J.: Single path one-shot neural architecture search with uniform sampling. arXiv preprint arXiv:1904.00420 (2019)
48. Cai, H., Zhu, L., Han, S.: Proxylessnas: Direct neural architecture search on target task and hardware. arXiv preprint arXiv:1812.00332 (2018)