SoftwareX 17 (2022) 100957

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Symbolic DNN-Tuner: A Python and ProbLog-based system for optimizing Deep Neural Networks hyperparameters

Michele Fraccaroli [a,*], Evelina Lamma [a], Fabrizio Riguzzi [b]

[a] Department of Engineering, University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy
[b] Department of Mathematics and Computer Science, University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy

ARTICLE INFO

Article history:
Received 28 September 2021
Received in revised form 14 December 2021
Accepted 14 December 2021

Keywords:
Deep learning
Probabilistic Logic Programming
Hyper-parameters tuning
Neural-symbolic integration

ABSTRACT

The application of deep learning models to increasingly complex contexts has led to a rise in the complexity of the models themselves. Due to this, there is an increase in the number of hyper-parameters (HPs) to be set and Hyper-Parameter Optimization (HPO) algorithms occupy a fundamental role in deep learning. Bayesian Optimization (BO) is the state-of-the-art of HPO for deep learning models. BO keeps track of past results and uses them to build a probabilistic model, building a probability density of HPs. This work aims to improve BO applied to Deep Neural Networks (DNNs) by an analysis of the results of the network on training and validation sets. This analysis is obtained by applying symbolic tuning rules, implemented in Probabilistic Logic Programming (PLP). 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, leading to improved performance. In this paper, we present the general system and its implementation. We also show its graphical interface and a simple example of execution.

Code metadata

Current code version	v2.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00183
Legal Code License	MIT License
Code versioning system used	Git
Software code languages, tools, and services used	Python, ProbLog, TensorFlow, Numpy, OpenCV, Scikit Optimize, Dash.
Compilation requirements, operating environments & dependencies	Pythion v3.6 or higher, TensorFlow v2.x or higher
Support email for questions	michele.fraccaroli@unife.it

1. Motivation and significance

With the increase in the complexity of Deep Neural Networks (DNNs), there is an increase in the number of hyper-parameters (HPs) to be set. But DNNs are very sensitive to the tuning of their HPs. Incorrect values of some of its HPs (i.e., learning rate or batch size) can make the difference between good and bad training (and consequently between good and bad networks).

Due to these reasons, Hyper-Parameters Optimization (HPO) algorithms gained more and more attention from researchers. Another way to tune DNNs HPs is to analyze the performance of the network (in terms of accuracy, precision, loss, etc.). From this analysis, it is possible to identify actions that can be applied to choosing HP values in order to obtain a network with better performance.

There are different HPO algorithms, and we can classify them into different categories. There are *exhaustive search* algorithms like Grid Search, *evolutionary* algorithms like Genetic Algorithms [1] and *Sequential Model-Based Optimization* (SMBO) [2] algorithms like Bayesian Optimization (BO) [3–5]. The algorithms in the first category perform a brute-force search and guarantee to find the optimal solution. The problem with these algorithms is that they suffer from the curse of dimensionality (the number of configurations to try is exponential w.r.t. the number of HPs to

* Corresponding author.
E-mail address: michele.fraccaroli@unife.it (Michele Fraccaroli).

https://doi.org/10.1016/j.softx.2021.100957

optimize). Genetic Algorithms can be a good idea for HPs tuning but, they are slow to converge and they do not guarantee to find the optimal solution. SMBO algorithms like BO are state-of-the-art in the task of deep learning HPs tuning. Specifically, BO uses a surrogate model such as an approximation of the objective function to optimize and try to fit previous experiments to identify where the local minimum might be. BO consists of two components: the *probabilistic regression model* (e.g., Gaussian Processes) and the *acquisition function*. The first component provides a posterior probability distribution that catches the uncertainty in the surrogate model and the *acquisition function* selects the next point to evaluate by measuring the value of the objective function at the new point based on the posterior distribution [3]. The success of BO in optimizing the HPs of DNNs is due to the fact that it limits the number of training of DNNs spending more time choosing the next set of HPs to try. In literature, there are different works that apply this type of optimization to DNNs HPs [5–7].

This work is about a software called Symbolic DNN-Tuner, that drives the training of DNNs, analyzes the performance of each DNN's training experiment and automatizes the choice of HPs values in order to improve the network's performance as much as possible. To do this, Symbolic DNN-Tuner exploits both manual approaches obtained with network performance analyses and an SMBO algorithm. In particular, in this work, BO is used as the SMBO algorithm.

BO was chosen because it limits the number of evaluations of the objective function by spending more time choosing the next set of HPs values to try. This is a perfect approach for DNNs. Given that the training phase is very expensive both in terms of time and energy, BO builds and uses a surrogate model of the objective function and quantifies the uncertainty in this surrogate using a regression model. In the end, it uses an acquisition function to decide where to sample the next set of HPs values [3–6,8].

The goal of Symbolic DNN-Tuner is to provide software that can drive the training phase of DNNs given only a neural network architecture, the HPs search space and the dataset.

In the rest of the paper, we show the software description explaining its architecture and functionality in Section 2, an illustrative example in Section 3 and the impact of the software in Section 4.

For the experimental evaluation on literature and production datasets, as evidence of performance improvements compared to the BO alone, see [9].

2. Software description

As mentioned before, Symbolic DNN-Tuner drive the training of DNNs analyzing the performance of each training and automatizes the choice of HPs values. The tricks used in manual approaches are mapped into non-deterministic and probabilistic Symbolic Tuning Rules (STRs). Each STRs identifies a Tuning Action (TA), which has the aim of updating the HPs search space or updating the network architecture. Each STR has a probabilistic weight that determines the probability of application of its TA to the case that the associated problem is diagnosed. The aim of STRs is to avoid typical DNNs problems like overfitting, underfitting, incorrect values of the learning rate, etc.

Symbolic DNN-Tuner has two main blocks: the *Neural Component* and the *Symbolic Component*. The whole Neural Component was developed in Python and the Symbolic Component in the probabilistic logic language ProbLog[1] [10]. In the Neural Component, the TensorFlow Python library was used for implementing and working with the network architecture, for running training and validation and for the input pipeline. For BO, we have used the Scikit-Optimize Python library.

The Neural Component manages the neural network, the HPs search space, BO and the application of the TAs. The Symbolic Component performs an analysis of the network's metrics after the training and validation phases, diagnoses problems and identifies the most probable TAs to be applied to the network architecture or HPs search space. Initially, probabilistic weights of STRs are set manually, and then they are refined, after each training, via Learning From Interpretations (LFI) [11] (an inductive algorithm available in PLP) based on the results obtained so far.

Table 1
Possible problems.

Behavior	Network 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

2.1. DNNs hyper-parameters, training problems and countermeasures

Each analysis performed on DNN's performance is associated with a possible diagnosis. This analysis is wholly implemented in Prolog. After retrieving the diagnosis of the DNN's problems, Symbolic DNN-Tuner has a set of TAs associated with the diagnosis that one can take. To address the problem identified in the analyses phase, the best TA to be applied to the network architecture or the HPs search space is identified by the ProbLog program. After the selection of the best TA, this is passed to the *Neural Component* who will take care of patching the neural network architecture or the HPs search space.

Tables 1 and 2 show the list of considered problems and the corresponding analyses, and the association of the problem with its TA respectively. In other words, the "Problem" column of Table 1 is the set of problems that Symbolic DNN-Tuner is able to detect. Table 3 reports the HPs that can be tuned and their domains.

2.2. Software architecture

As mentioned, Symbolic DNN-Tuner is composed of two main parts: the *Neural Component* and the *Symbolic Component*. The *Symbolic Component* receives the network performance in terms of metrics like loss and accuracy from *Neural Component*, performs symbolic analyses and chooses the TAs to be applied to the network. These TAs are taken by the *Neural Component* and applied to the DNN architecture and/or HPs space. A schematic representation of the execution pipeline and the two main blocks of Symbolic DNN-Tuner are shown in Fig. 1.

The two parts of the software communicate thanks to the fact that ProbLog is developed in Python and is usable as a Python package that can be called directly in Python code.

Symbolic DNN-Tuner is composed of various software modules as shown in Fig. 2. The first is the **Main** module. It takes the data, the definition of the HPs search space and manages BO. The core module is **Controller**. It takes the data from the Main module and manages the pipeline of starting the training of the neural network (including the validation phase), the symbolic analysis, LFI, and the application of selected TAs. **Neural**, **Diagnosis** and **Tuning** are stand-alone modules. **Neural** takes care of

[1] ProbLog website: https://dtai.cs.kuleuven.be/problog/.

Table 2
TA associated to the problems.

Problem	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

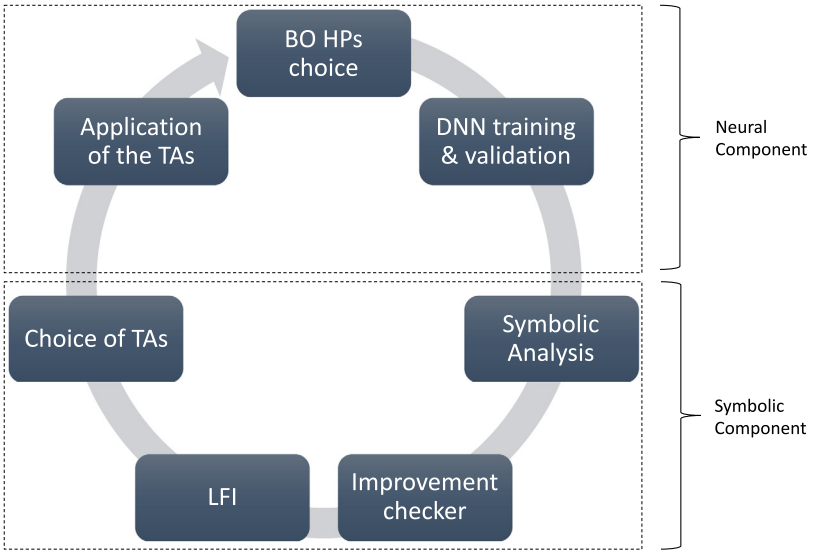


Fig. 1. Symbolic DNN-Tuner execution pipeline with both Neural block and Symbolic block.

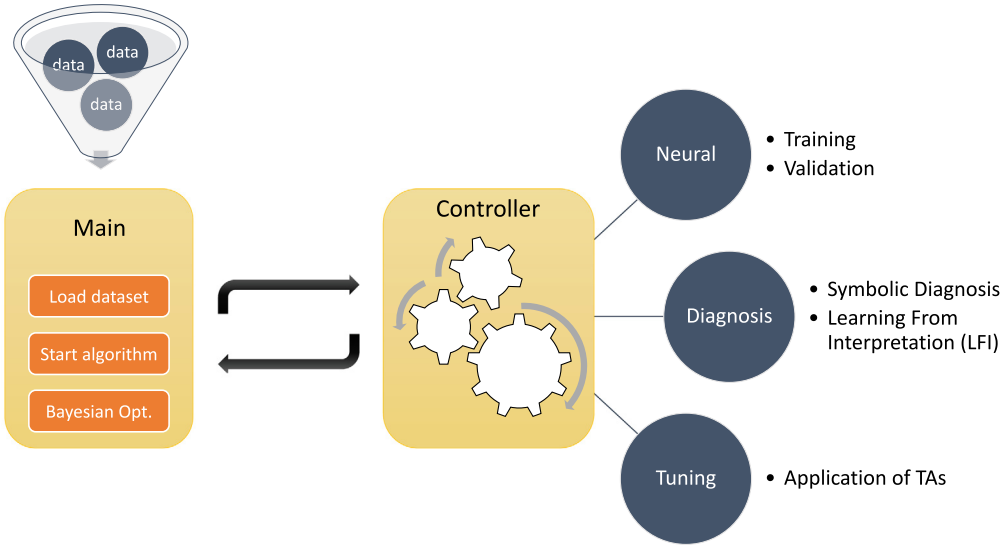
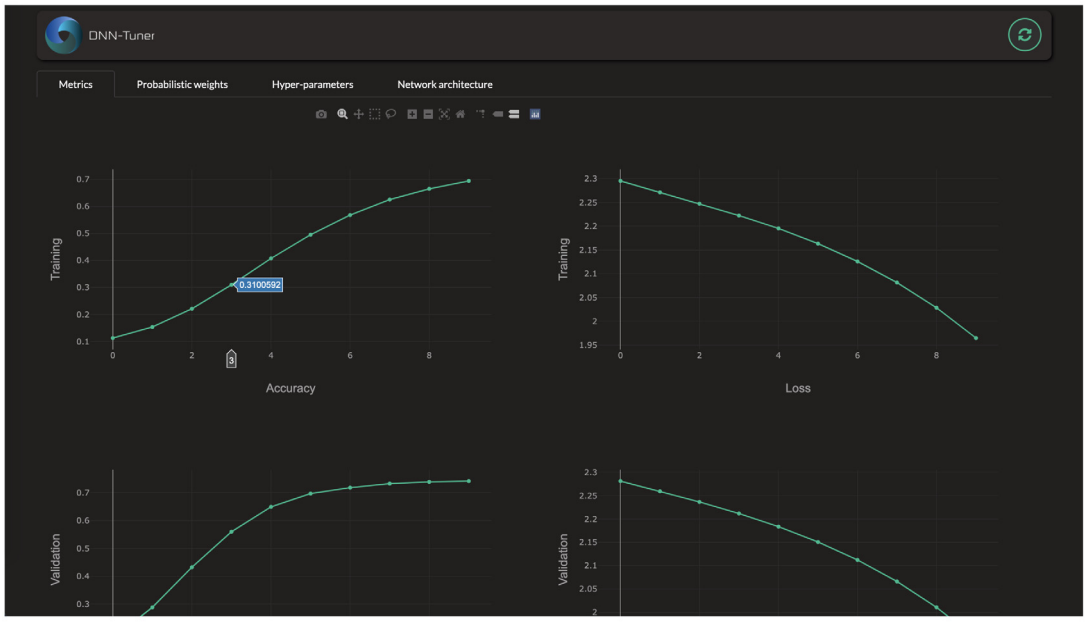
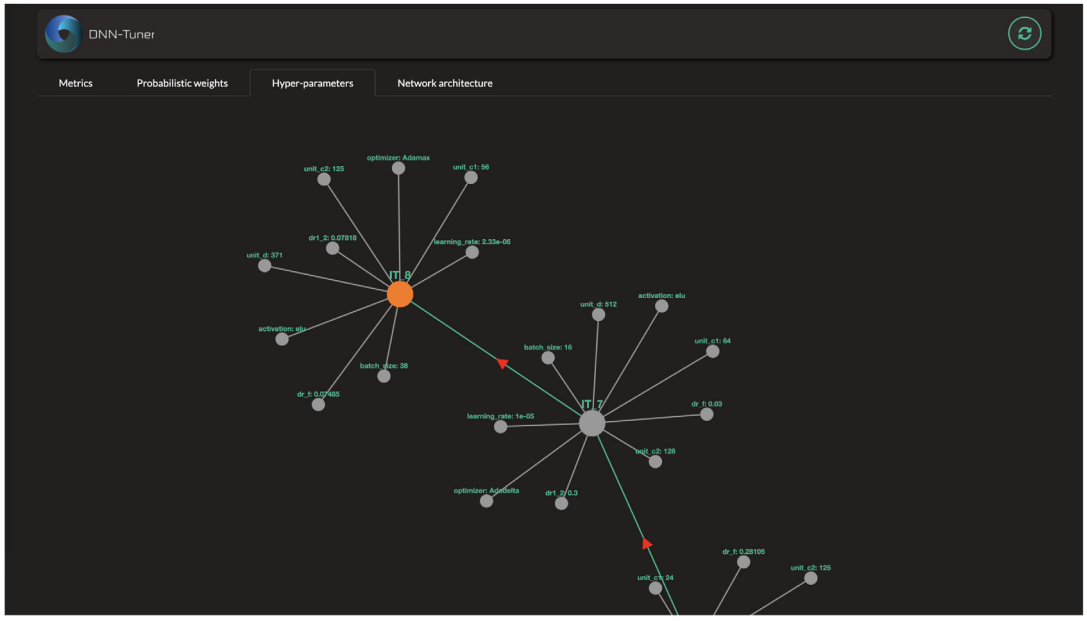


Fig. 2. Software modules of Symbolic DNN-Tuner.

the managing of the DNN. After training, it performs validation and applies the structural changes to the neural network. The **Diagnosis** module is fully written in ProbLog and deals with the analysis of the performance of the neural network, identifying



(a) Metrics of current training



(b) Hyper-parameters

Fig. 3. Symbolic DNN-Tuner's dashboard for monitoring the whole optimization process: (a) page with the metrics, (b) page with the HPs used in each iteration of the software.

Table 3
Hyper-Parameters (HPs).

Hyper-Parameter	Domain
First convolutional layer	16–64 filters
Second convolutional layer	64–128 filters
First dropout	0.002–0.3
Second dropout	0.03–0.5
Fully connected layer	256–512 neurons
Learning rate	10^{-5}–10^{-1}
Activation function	[ReLU, ELU, SELU]
Optimizer	[Adam, Adamax, RMSprop, Adadelta]

problems. The Diagnosis module also applies LFI and chooses the most appropriate TA to apply. **Tuning** applies the TAs and updates the HPs search space or tells Neural what to change in the network's architecture.

2.3. Software functionalities

The main functionality of Symbolic DNN-Tuner is to drive the training of DNNs to obtain as far as possible the network with the best performance. The optimization process exploits the analysis of the network's performance after each training and validation phase. After the analysis, LFI [11] is used to learn the probability of the tuning rules and therefore dynamically change the probabilistic logic program. This allows Symbolic DNN-Tuner to favor the application of the rules that were more effective in improving the performance of DNNs.

2.3.1. Dataset definition

Symbolic DNN-Tuner has a separate script dedicated to the dataset definition. In this script, each dataset is a function that returns the dataset, divided into training and test set, and the number of classes. Listing 1 shows an example of the definition of the CIFAR10 [12] dataset.

Listing 1: CIFAR10 Dataset Definition

```
from tensorflow.keras.datasets import cifar10
def cifar10_dataset():
        num_classes = 10
        # The data, split between train and test sets:
        (x_train, y_train), (x_test, y_test) = cifar10
            .load_data()
        print(x_train.shape[0], 'train samples')
        print(x_test.shape[0], 'test samples')

        # Convert class vectors to binary class
            matrices.
        y_train = tf.keras.utils.to_categorical(
            y_train, num_classes)
        y_test = tf.keras.utils.to_categorical(y_test,
             num_classes)

        return x_train, x_test, y_train, y_test,
            num_classes
```

Now, in the **Main** module, it is possible to obtain the split dataset and the number of classes by simply calling `cifar10_dataset()`. Through this script, it is possible to add any dataset that respects the output interface described in Listing 1. In future work, we will enable the use of data generators to manage large amounts of data in an optimized way.

2.3.2. Tracking and monitoring of experiments

Symbolic DNN-Tuner has a dashboard developed with the Dash framework.[2] and Netron[3,4] Figs. 3 and 4 show the four pages of the dashboard. During the execution, all events are logged. All trained models, together with their weights, the diagnosis, the TAs applied, are saved. The Neural Component can also log the training information to be monitored using TensorBoard[5] if more details are needed on the performance of the networks.

Fig. 3 shows the tabs of metrics and HPs of the dashboard. Fig. 3(a) displays metrics such as accuracy and loss during the training and validation phases. Fig. 3(b) shows the HPs used in each iteration of the software. The orange point is the last iteration with its HPs. To see the whole history of the HPs used in each iteration of Symbolic DNN-Tuner, you can follow the little red arrow in the HPs graph.

Fig. 4 shows the tabs of probabilistic weights and network architecture of the dashboard. Fig. 4(a) shows the page with the probabilistic weights of the STRs. At each iteration of the Symbolic DNN-Tuner, it is possible to see which TA was found to be more effective during the HP optimization process. Fig. 4(b) shows the integration with Netron for visualizing the network architecture. Thanks to the integration with Netron, it is possible to interact with the network and see more details for each layer (e.g., activation, kernel size, kernel regularization, etc.).

[2] Dash framework webpage: https://plotly.com/dash/.

[3] Netron webpage: https://lutzroeder.github.io/netron/.

[4] Netron GitHub repo: https://github.com/lutzroeder/netron.

[5] TensorBoard website: https://www.tensorflow.org/tensorboard.

3. Illustrative examples

We now discuss an example of training performed on the CIFAR10 [12] dataset. The dataset and the HP search space are defined as in listings 2 and 3 respectively. Then the software is ready to start the training of the neural network. Executing `python3 main.py` starts Symbolic DNN-Tuner.

With the script for the definition of the dataset, it is possible to import CIFAR10. As described in Section 2.3.1, the Main module takes the split dataset returned by this function and the number of classes of the dataset. These variables are passed to the Controller. The Controller module passes them to the Neural module which uses them to define the last layer of the neural network and to perform the training and validation of the model. Listing 3 shows the definition of the HPs search space. Each element of the list `self.search_space` refers to a specific HP in accordance with the definition of the neural network (Listing 4).

BO initially uses random values from the HPs search space defined in Listing 3 with the neural network objective function to optimize. BO is implemented by the library Scikit-Optimize and it takes as input a function to minimize. This function also takes as input a list of parameters (network HPs) and returns a single value of the objective function (in this case the value of loss function), see Listing 5.

After running the `main.py` script, by simply launching TensorBoard on the logging directory at path `log_folder/logs/`, it is possible to see the status of the training of the network. Alternatively, by executing `python3 dashboard/launcher.py`, it is possible to run the custom dashboard and monitor the status of the training, the weights of the TAs and the HPs used at each training (see Section 2.3.2).

Listing 2: dataset loading

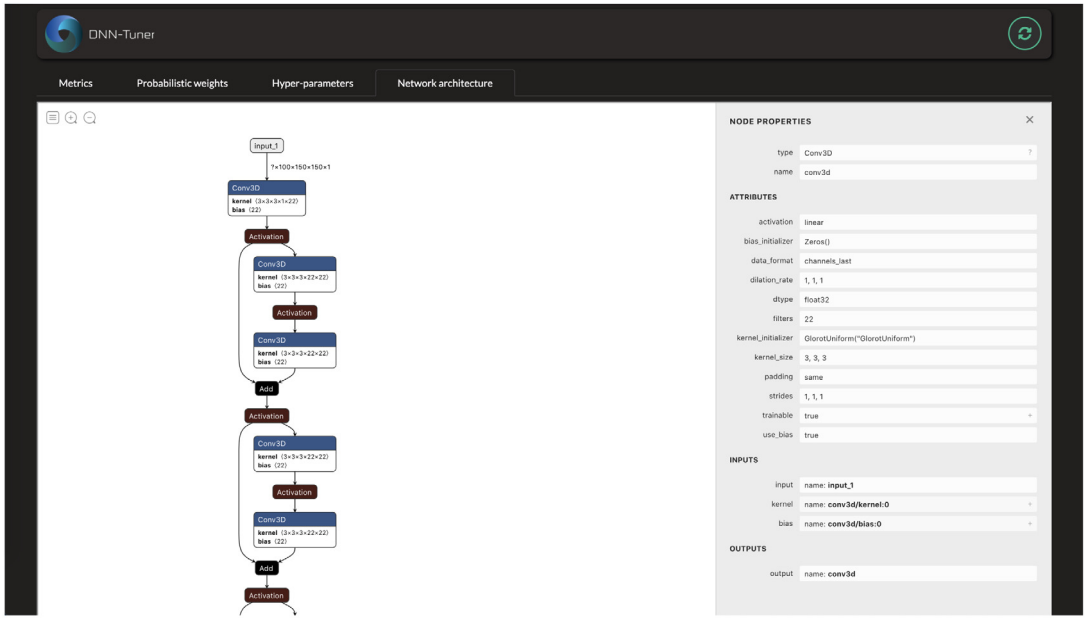
```
from tensorflow.keraas.dataset import cifar10
def mnist():
        num_classes = 10
        (x_train, y_train), (x_test, y_test) = cifar10
            .load_data()
        y_train = tf.keras.utils.to_categorical(
            y_train, num_classes)
        y_test = tf.keras.utils.to_categorical(y_test,
             num_classes)
        x_train = x_train.reshape(x_train.shape[0],
            32, 32, 3)
        x_test = x_test.reshape(x_test.shape[0], 32,
            32, 3)
        return x_train, x_test, y_train, y_test,
            num_classes
```

Listing 3: HPs search space definition

```
from skopt.space import Integer, Real, Categorical
def search_sp(self):
        self.search_space = [
        Integer(16, 64, name='unit_c1'),
        Real(0.002, 0.3, name='dr1_2'),
        Integer(64, 128, name='unit_c2'),
        Integer(256, 512, name='unit_d'),
        Real(0.03, 0.5, name='dr_f'),
        Real(1e-4, 1e-3, name='learning_rate'),
        Integer(16, 256, name='batch_size'),
        Categorical(['Adam', 'Adamax', 'Adadelta'],
            name='optimizer'),
        Categorical(['relu', 'elu', 'selu'], name='
            activation')
        ]

        return self.search_space
```

(a) Probabilistic Weights



(b) Network architecture

Fig. 4. Symbolic DNN-Tuner's dashboard: (a) page with the probabilistic weights of the STRs of the Symbolic Parts, (b) visualization of the network's architecture using Netron.

Listing 4: Neural network architecture definition through Keras Functional API

```
def build_network(self, params, new):
        """
        Function for define the network structure
        :return: model
        """
        inputs = Input((32, 32, 3))

        x = Conv2D(params['unit_c1'], (3, 3), padding
            ='same')(inputs)
        x = Activation(params['activation'])(x)
        x = Conv2D(params['unit_c1'], (3, 3))(x)
        x = Activation(params['activation'])(x)
        x = MaxPooling2D(pool_size=(2, 2))(x)
        x = Dropout(params['dr1_2'])(x)

        x = Conv2D(params['unit_c2'], (3, 3), padding
            ='same')(x)
        x = Activation(params['activation'])(x)
        x = Conv2D(params['unit_c2'], (3, 3))(x)
        x = Activation(params['activation'])(x)
        x = MaxPooling2D(pool_size=(2, 2))(x)
        x = Dropout(params['dr1_2'])(x)

        x = Flatten()(x)
        x = Dense(params['unit_d'])(x)
        x = Activation(params['activation'])(x)
        x = Dropout(params['dr_f'])(x)
```

```
        x = Dense(self.n_classes)(x)
        x = Activation('softmax')(x)

        model = Model(inputs=inputs, outputs=x)
        return model
```

Listing 5: BO implementation

```
def objective(params):
        to_optimize = controller.training(space)
        return to_optimize

search_res = gp_minimize(objective, search_space,
    acq_func='EI', n_calls=1, n_random_starts=1,
    callback=[checkpoint_saver])
```

4. Impact

We envision the application of this software to each experiment that concerns DNNs. This software can be used by both experienced and inexperienced deep learning users. This is because Symbolic DNN-Tuner only needs the dataset, the HP search space and the neural network definition. Once these three elements have been defined, Symbolic DNN-Tuner will take care of modifying the network to try to obtain the best possible performance. Moreover, thanks to the neural-symbolic integration obtained by the exploitation of Probabilistic Logic Programming, this software can be used to obtain a form of explanation of the possible reasons for network malfunctioning.

This software has been tested not only on the classic benchmark datasets but also on a dataset provided by CIMA S.P.A.[6] This dataset was used to test Symbolic DNN-Tuner on an industrial, real case [9].

5. Conclusions

We have presented Symbolic DNN-Tuner, a system for automatically driving the training of DNNs, by automatizing the choice of hyper-parameters. This automation was achieved by combining BO with an analysis of the network's performance implemented by rule-based programming. In particular, tuning rules have been implemented in Probabilistic Logic Programming and their weights are tuned by exploiting Learning From Interpretation. Thus the software exploits probabilistic symbolic rules for selecting after each network training, the best tuning action to correct the identified problems. These tuning actions update the network architecture and/or update the hyper-parameters search space.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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

References

[1] Whitley D. A genetic algorithm tutorial. Stat Comput 1994;4(2):65–85.
[2] Bergstra JS, Bardenet R, Bengio Y, Kégl B. Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems. 2011, p. 2546–54.
[3] Frazier PI. A tutorial on Bayesian optimization. 2018, arXiv preprint arXiv:1807.02811.
[4] Dewancker I, McCourt M, Clark S. Bayesian optimization for machine learning: A practical guidebook. 2016, arXiv preprint arXiv:1612.04858.
[5] Snoek J, Larochelle H, Adams RP. Practical Bayesian optimization of machine learning algorithms. In: Advances in neural information processing systems. 2012, p. 2951–9.
[6] Guillemot M, Heusèle C, Korichi R, Schnebert S, Petit M, Chen L. Tuning neural network hyperparameters through Bayesian optimization and application to cosmetic formulation data. In: ORASIS 2019. 2019.
[7] 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.
[8] Pelikan M, Goldberg DE, Cantú-Paz E. BOA: The Bayesian optimization algorithm. In: Proceedings of the 1st annual conference on genetic and evolutionary computation-Vol. 1. Morgan Kaufmann Publishers Inc.; 1999, p. 525–32.
[9] Fraccaroli M, Lamma E, Riguzzi F. Symbolic DNN-tuner. Mach Learn 2021;1–26.
[10] De Raedt L, Kimmig A, Toivonen H. Problog: A probabilistic prolog and its application in link discovery.. In: IJCAI, vol. 7. Hyderabad; 2007, p. 2462–7.
[11] 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. Springer; 2011, p. 581–96.
[12] Krizhevsky A, Nair V, Hinton G. Cifar-10. Canadian Institute for Advanced Research URL http://www.cs.toronto.edu/~kriz/cifar.html.

[6] CIMA website: http://www.cima-cash-handling.com/it/.