Automatic Setting of DNN Hyper-Parameters by Mixing Bayesian Optimization and Tuning Rules

Michele Fraccaroli[1][0000−0001−7656−7204], Evelina Lamma[1][0000−0003−2747−4292], and Fabrizio Riguzzi[2][0000−0003−1654−9703]

[1] DE - Department of Engineering
[2] DMI - Department of Mathematics and Computer Science
University of Ferrara
Via Saragat 1, 44122 Ferrara, Italy
{michele.fraccaroli, evelina.lamma, fabrizio.riguzzi}@unife.it

Abstract. Deep learning techniques play an increasingly important role in industrial and research environments due to their outstanding results. However, the large number of hyper-parameters to be set may lead to errors if they are set manually. The state-of-the-art hyper-parameters tuning methods are grid search, random search, and Bayesian Optimization. The first two methods are expensive because they try, respectively, all possible combinations and random combinations of hyper-parameters. Bayesian Optimization, instead, builds a surrogate model of the objective function, quantifies the uncertainty in the surrogate using Gaussian Process Regression and uses an acquisition function to decide where to sample the new set of hyper-parameters. This work faces the field of Hyper-Parameters Optimization (HPO). The aim is to improve Bayesian Optimization applied to Deep Neural Networks. For this goal, we build a new algorithm for evaluating and analyzing the results of the network on the training and validation sets and use a set of tuning rules to add new hyper-parameters and/or to reduce the hyper-parameter search space to select a better combination.

1 Introduction

Deep Neural Networks (DNNs) provide outstanding results in many fields but, unfortunately, they are also very sensitive to the tuning of their hyper-parameters. Automatic hyper-parameters optimization algorithms have recently shown good performance and, in some cases, results comparable with human experts [3]. Tuning a big DNN is computationally expensive and some experts still perform manual tuning of the hyper-parameters. To do this, one can refer to some tricks [15] to determine the best set of hyper-parameter values to use for obtaining good performance from the network. This work aims to combine the automatic approach with these tricks used in the manual approach but in a fully-automated integration for drive the training of DNNs, automatizing the choice of HPs and analyzing the performance of each training experiment to obtain a network with better performance. Heuristics and Bayesian Optimization for parameter tuning have already been used in other application contexts other than Deep Learning, e.g., using heuristics to shrink the size of a parameter space for Self-Adapting Numerical Software (SANS) systems [6], using rollout heuristics that work well with Bayesian optimization when a finite budget of total evaluations is prescribed [14] or using domain-specific knowledge to reduce the dimensionality of Bayesian optimization to tune a robot's locomotion controllers [16]. For the automatic approach, we use a Bayesian Optimization [5] because it limits the evaluations of the objective function (training and validation of the neural network in our case) by spending more time in choosing the next set of hyper-parameter values to try. We aim at improving Bayesian Optimization [5] by integrating it with a set of tuning rules. These rules have the purpose of reducing the hyper-parameter space or adding new hyper-parameters to set. In this way, we constrain the Bayesian approach to select the hyper-parameters in a restricted space and add new parameters without any human intervention. In this way, we can avoid typical problems of the neural networks like overfitting and underfitting, and automatically drive the learning process to good results.

After discussing Bayesian Optimization (Section 2), we introduce the approach used to analyze the behaviour of the networks, the execution pipeline, the performed analysis and the tuning rules used to fix the detected problems (Section 3, Section 3.1, Section 3.2, and Section 3.3). Experimental results with different networks and different datasets are described in Section 4. All experiments are performed on benchmark datasets, MNIST and CIFAR-10 in particular.

2 Bayesian Optimization

In this section, we shortly review the state-of-the-art of hyper-parameter optimization for DNNs, i.e., grid search, random search and, with special attention, Bayesian Optimization. Grid Search applies exhaustive research (also called *brute-force search*) through a manually specified hyper-parameter space. This search is guided by some specified performance metrics. Grid Search tests the learning algorithm with every combination of the hyper-parameters and, thanks to its *brute-force search*, guarantees to find the optimal configuration. But, of course, this method suffers from the *curse of dimensionality*. The problem of Grid Search applied to DNNs is that we have a huge amount of hyper-parameters to set and the evaluating phase of the algorithm is very expensive. This definitively raises a time problem.

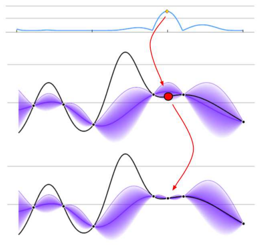
Random Search uses the same hyper-parameter space of Grid Search, but replaces the *brute-force search* with *random search*. It is, therefore, possible that this approach will not find results as accurate as Grid Search, but it requires less computational time while finding a reasonably good model in most cases [2].

Bayesian Optimization (BO) is a probabilistic model-based approach for optimizing objective functions (f) which are very expensive or slow to evaluate [5].

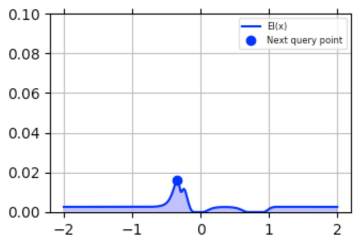
The key idea of this approach is to limit the evaluation phase of the objective function by spending more time to choosing the next set of hyper-parameters' values to try. In particular, this method is focused on solving the problem of finding the maximum of an expensive function $f : X \to \mathbb{R}$,

$$argmax_{x \in X} f(x) \tag{1}$$

where X is the hyper-parameter space while can be seen as a hyper-cube where each dimension is a hyper-parameter. BO builds a surrogate model of the objective function and quantifies the uncertainty in this surrogate using a regression model (e.g., *Gaussian Process Regression*, see next section). Then it uses an acquisition function to decide where to sample the next set of hyper-parameter values [7]. The two main components of Bayesian Optimization are: the *statistical model* (regression model) and the *acquisition function* for deciding the next sampling. The statistical model provides a posterior probability distribution that describes potential values of the objective function at candidate points. As the number of observations grows, the posterior distribution improves and the algorithm becomes more certain of which regions in hyper-parameter space are worth to be explored and which are not. The acquisition function is used to find the point where the objective function is supposed to be maximal [12]. This function defines a balance between *Exploration* (new areas in the parameter space) and *Exploitation* (picking values in areas that are already known to have favorable values); the aim of the *Exploration* phase is to select samples that shrink the search space as much as possible, and the aim of the *Exploitation* phase is to focus on the reduced search space and to select samples close to the optimum [10]. Figure 1 (a) shows the behaviour of this algorithm.



(a)



(b)

Fig. 1: (a) BO behaviour. The figure shows the objective function (black line), mean (light purple line), covariance (purple halo) and the samplings (black dots). The top of the figure shows the Expected Improvement function (blue line) and his maximum (yellow dot). The next sampled point will be exactly the maximum of the activation function (red dot). The bottom of the figure shows the new statistical model [4]. (b) Example of Expected Improvement with the sampling of the next point (blue dot) generated with Scikit-Optimize library.

A Gaussian Process (GP) is a stochastic process and a powerful prior distribution on functions. Any finite set of N points $\{x_n \in X\}_{n=1}^N$ induces a multivariate Gaussian distribution on $\mathbb{R}^N$. GP is completely determined by the mean function $m : X \to \mathbb{R}$ and the covariance function $K : X \times X \to \mathbb{R}$ [17]. For a complete and more precise overview of the Gaussian Process and its application to Machine Learning, see [17].

The acquisition function is used to decide where to sample the new set of hyper-parameters' values, Figure 1(b) shows. In the literature, there are many types of acquisition function. Usually, only three are used: *Probability of Improvement*, *Expected Improvement* and *GP Upper Confidence Bound* (GP-UCB) [12] [18]. The acquisition function used in this work is *Expected Improvement* [11]. The idea behind this acquisition function is to define a non-negative expected improvement over the best previously observed target value (previous best value) at a given point x. Expected Improvement is formalized as follows:

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x)\, dy \tag{2}$$

where y^* is the actual best result of the objective function, x is the new proposed set of hyper-parameters' values, y is the actual value of the objective function using x as hyper-parameters' values and $p(y|x)$ is the surrogate probability model expressing the probability of y given x.

3 Network Behaviour Analysis

By automatically analyzing the behaviour of the network, it is possible to identify problems that DNNs could have after the training phase (e.g., overfitting, underfitting, etc). Bayesian Optimization only works with a single metric (validation loss or accuracy, training loss or accuracy) and is not able to find problems like overfitting or fluctuating loss. When these problems are diagnosed, the proposed algorithm aims to shrink the hyper-parameters' value space or update the network architecture to drive the training to a better solution. Many types of analyses can be performed on the network results, e.g., analyzing the relation between training and validation loss or the accuracy, the trend of accuracy or loss during the training phase in terms of direction and form of the trend. For example: if a significant difference between the training loss and validation loss is found, and the validation loss is greater then the training loss, the diagnosis is that the network has a high variance and overfitting occurred.

3.1 Execution Pipeline

The software structure is composed by four main modules: *training* module (NN in Figure 2), *Diagnosis* module, *Tuning Rules* module and the *Search Space* driven by the *controller* as can be seen in Figure 2. Controller rules the whole process. The main loop of the Algorithm 1 uses the Controller for running the

Diagnosis module, the *Tuning Rules* module and BO for performing the optimization. BO is used for choose the hyper-parameters' value and is forced to do so from a restricted search space.

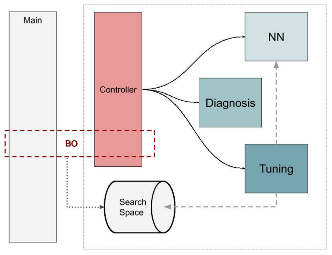


Fig. 2: The structure of the software with the main modules.

Algorithm 1 encapsulates the whole process (called *DNN-Tuner*). With S, N, $Params$ and n we refer respectively to the hyper-parameters' value search space, initial neural network definition, a value picked from the search space and the number of cycles of the algorithm. R, H, and M are the results of the evaluation of the trained network, the history of the loss and accuracy in both training and validation phases and the trained model of the network, respectively. $Ckpt$ is a checkpoint of the Bayesian Algorithm that can be used to restore the state of the Bayesian Optimization. $Issues$, $NewM$ and are a list of issues found by the *Diagnosis* module, the new model, respectively. All the modules are implemented in Python. The DNN and BO are implemented with Keras and Scikit-Optimize respectively.

Algorithm 1 DNN-Tuner

```
Require: $S$, $N$, $n$
  $Params \leftarrow Randomization(S)$
  $M \leftarrow BuildModel(P, N)$
  $R, H \leftarrow Training(Params, M)$
  $Ckpt \leftarrow None$
  while $n > 0$ do
    $Issues \leftarrow Diagnosis(H, R)$
    $NewM, S \leftarrow Tuning(Issues, M, S)$
    if $NewM \neq M$ then
      $Params \leftarrow Randomization(S)$
      $R, H \leftarrow Training(Params, NewM)$
    else
      $Ckpt \leftarrow BO(S, NewM, Ckpt)$
    end if
    $n \leftarrow (n - 1)$
  end while
```

3.2 Diagnosis Module

We have performed some proof-of-concept analyses. DNN-Tuner analyses the correlation between training and validation accuracy and loss to detect overfitting. It also analyses validation loss and accuracy to detect underfitting, and the trend of loss to fix the learning rate and the batch size.

For the first analysis, the proposed algorithm evaluates the difference of the accuracy in the training and validation phases to identify any gap as shown by Figure 3. An important gap between these two plots is a clear sign of overfitting since this means that the network classifies well training data but not so well validation data, that is, it is unable to generalize. In this case, the algorithm can diagnose overfitting and then take action to correct this behaviour.

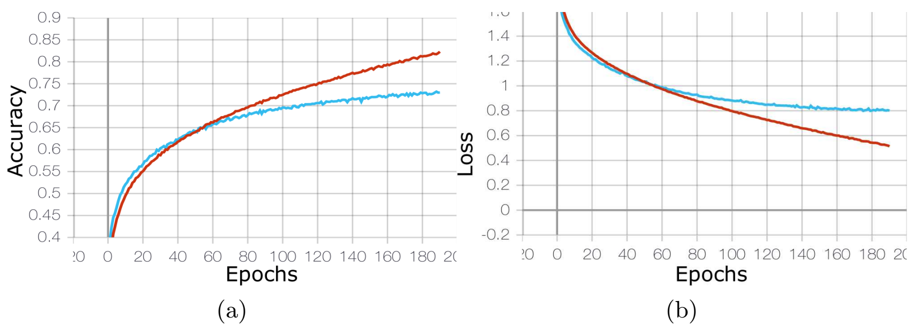


Fig. 3: An example of accuracy (a) and loss (b) in the training (red line) and validation phase (light blue line).

For detecting underfitting, DNN-Tuner analyses the results of loss in both phases and estimates the amount of loss. The loss value that can be used as a threshold to diagnose underfitting is variable and depends on the problem. Then, a threshold is imposed and, if this threshold is exceeded, DNN-Tuner will diagnose the underfitting.

The analysis of the trend of the loss is useful for detecting the effects of the learning rate and batch size on the network. Given the trend, the proposed algorithm can check whether the loss is too noisy, and whether the direction of the loss is descending, ascending or constant. In response to this diagnosis, the algorithm can attempt to set the batch size and learning rate correctly. These adjustments are performed by applying rules called *Tuning Rules* (Section 3.3).

3.3 Tuning Rules

Tuning rules are the actions applied to the network or to the hyper-parameters' value search space in response to the issues identified with the *Diagnosis* module. Then, for each issue found by the diagnosis module, a tuning rule is applied and a new search space and a new model are derived from the previous ones. In the context of this work, the rules applied for each issue are shown Table 1.

Issue	Tuning Rules
Overfitting	Regularization
	Batch Normalization
Underfitting	More neurons
Fluctuating loss	Increase of batch size
Increasing loss trend	Decrease of learning rate

Table 1: Rules applied on each issue.

In order to prevent overfitting, the tuning module applies Batch Normalization [9] and L2 regularization [13]. For underfitting, it tries to increase the learning capacity of the network by removing small values from the space of the number of neurons in the convolutional and the fully connected layers. In this way, it drives the optimization to choose higher values for the number of neurons. If DNN-Tuner finds the trend of the loss to be fluctuating, it forces the optimization to choose larger batch size values.

The amount of oscillation in the loss is related to the batch size. When the batch size is 1, the oscillation will be relatively high 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). The last rule of Table 1 tries to fix a growing loss. When the loss grows or remains constant at too high values, this means that the learning rate is probably too high and the learning algorithm is unable to find the minimum because it takes too large steps. When the algorithm detects this behaviour, it removes large values from the learning rate search space. Tuning is applied following the IF-THEN paradigm as shown in Algorithm 2, where $NewS$ is the altered search space.

Algorithm 2 Tuning

```
Require: $Issues, M, S$
  for $I$ in $Issue$ do
    if $I =$ "$overfitting$" then
      $NewM, NewS \leftarrow Apply([Regularization, Batch\ Normalization], M, S)$
    else if $I =$ "$underfitting$" then
      $NewM, NewS \leftarrow Apply(More\ neurons, M, S)$
      ...
    end if
  end for
  return $NewM, NewS$
```

4 Experiments

We compare *DNN-Tuner* with standard Bayesian Optimization implemented with the library Scikit-Optimize. Experiments were performed on CIFAR-10 and MNIST dataset. For each experiment, a cluster with NVIDIA GPU K80 and 8-cores Intel Xeon E5-2630 v3 was used. Due to the hardware settings, there is a time limit of 8 hours for each the experiment.

The initial state of the DNN is composed of two blocks consisting of two Convolutional layers with ReLU as activation followed by a MaxPooling layer. At the end of the second block, there is a Dropout layer followed by a chain of two Fully Connected layers separated by a Dropout layer. The initial hyper-parameters to be set by the algorithm are the number of the neurons in the Convolutional layers, the values of the Dropout layers, the learning rate and the batch size. The size of the search space depends on hyper-parameter. The domains of hyperparameters are as follows. The size of the first and second convolutional layers are between 16 and 48 and between 64 and128 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 layes is between 256 and 512. The domain of the learning rate is $[10^{-5}, 10^{-1}]$.

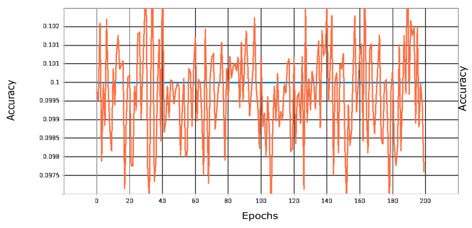
Unlike Bayesian Optimization, our algorithm can update the network structure and the hyper-parameters to be set. For example, when our algorithm detects overfitting, it updates the network structure by adding L2 regularization at each Convolutional layers (kernel regularization), Batch Normalization after the activations and adds L2 regularization parameters to the hyper-parameters to be set. Both DNN-Tuner and BO start with the same neural network and the same hyper-parameters to tune and the same domains for hyperparameters..

Figures 4 to 8 show accuracy and loss in training and validation phases of both algorithms (DNN-Tuner and BO). Both algorithms have completed seven optimization cycles. The neural network has been trained for 200 epochs and the dataset used are CIFAR-10 and MNIST.

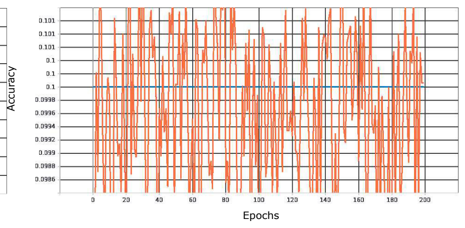
Figures 4 show the difference in training and validation at the first cycle of optimization of DNN-Tuner and Bayesian Optimization respectively. You can observe that the trends are noisy because, in the first cycle, the hyper-parameters' values are chosen randomly.

Figures 5 show the difference in training and validation at the fourth cycle of optimization of DNN-Tuner and Bayesian Optimization respectively. You can observe that the trends of BO continue to be noisy, while the DNN-Tuner trends become less fluctuating with an accuracy of ~ 0.75 and a loss of ~ 0.8 on the validation set.

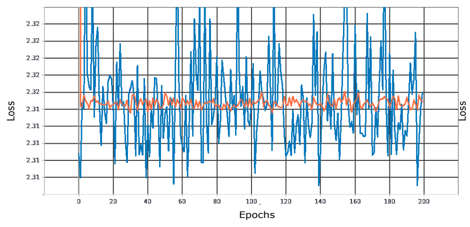
Figures 6 show the difference in training and validation at the seventh and last cycle of optimization of DNN-Tuner and Bayesian Optimization respectively. You can observe that the trends of BO still continues to be noisy, while the DNN-Tuner trends reach an accuracy of ~ 0.83 and a loss of ~ 2 on the validation set.



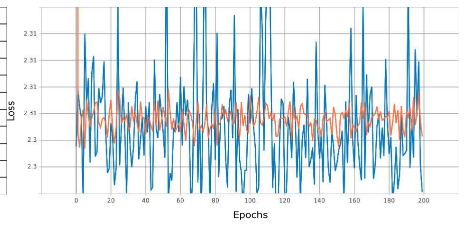
(a) Accuracy of DNN-Tuner.



(b) Accuracy of BO.

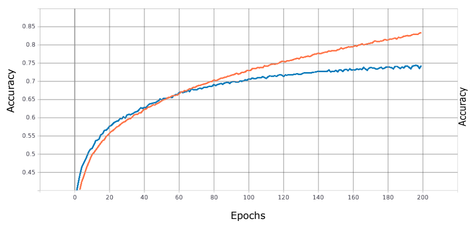


(c) Loss of DNN-Tuner.

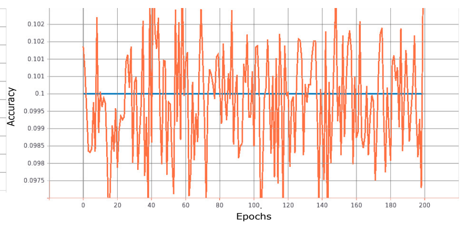


(d) Loss of BO.

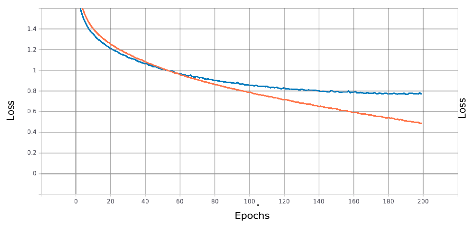
Fig. 4: Accuracy and loss in training (orange line) and validation (blue line) in the first step of both algorithms (DNN-Tuner and Bayesian Optimization).



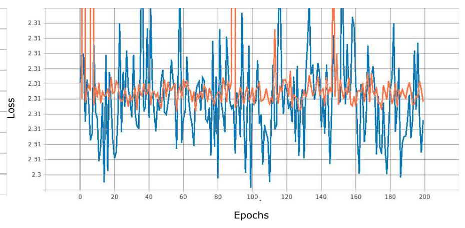
(a) Accuracy of DNN-Tuner.



(b) Accuracy of BO.

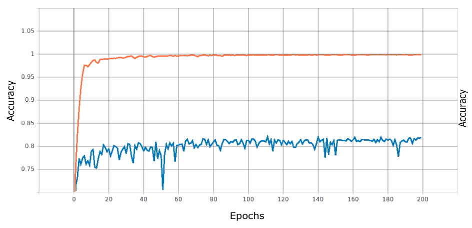


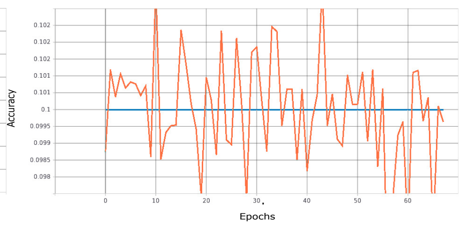
(c) Loss of DNN-Tuner.

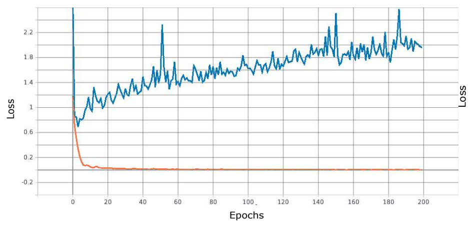


(d) Loss of BO.

Fig. 5: Accuracy and loss in training (orange line) and validation (blue line) in the fourth step of both algorithms.


(a) Accuracy of DNN-Tuner.


(b) Accuracy of BO.


(c) Loss of DNN-Tuner.

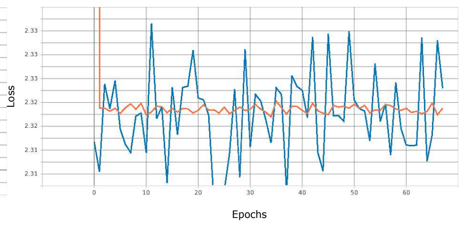
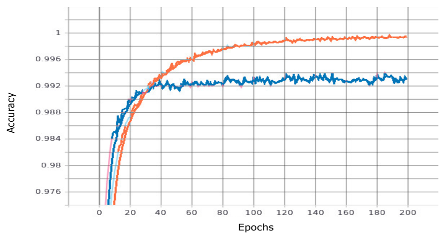

(d) Loss of BO.

Fig. 6: Accuracy and loss in training (orange line) and validation (blue line) in the seventh step of both algorithms.


(a) Accuracy of DNN-Tuner.

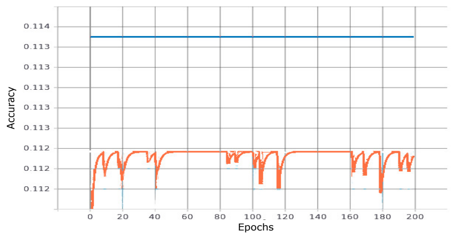
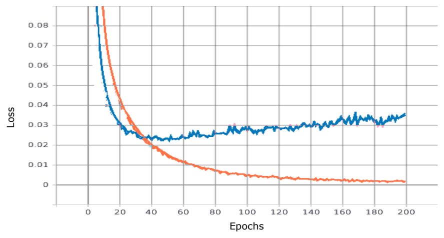

(b) Accuracy of BO.

Fig. 7: Accuracy and loss in training (orange line) and validation (blue line) of the seventh step of both algorithms.


(a) Loss of DNN-Tuner.

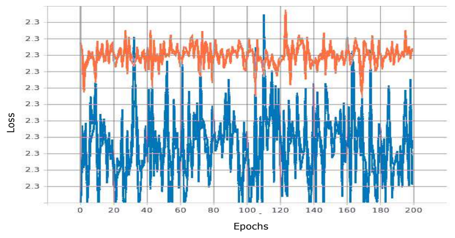

(b) Loss of BO.

Fig. 8: Accuracy and loss in training (orange line) and validation (blue line) of the seventh step of both algorithms.

Figures 7 and 8 show the difference of training and validation at the seventh and last cycle of optimization of DNN-Tuner and Bayesian Optimization on the MNIST dataset, also with 200 training epochs. We can see that DNN-Tuner, reaches an accuracy ~ 0.993 and a loss of ~ 0.04 in the validation set unlike BO which reaches ~ 0.114 and ~ 2.3. In terms of time, for the seven cycles of optimization with 200 training epochs on MNIST dataset: DNN-Tuner employed 4.75 hours while Bayesian Optimization 5.74 hours.

5 Conclusion and Future Work

DNNs have achieved extraordinary results but the number of hyper-parameters to be set has led to new challenges in the field of Artificial Intelligence. Given this problem and the ever-larger size of the networks, the standard approaches of hyper-parameters tuning like Grid and Random Search are no longer able to provide satisfactory results in an acceptable time. Inspired by the Bayesian Optimization (BO) technique, we have inserted expert knowledge into the tuning process, comparing our algorithm with BO in terms of quality of the results and time. The experiments show that it is possible to combine these approaches to obtain a better optimization of DNNs. The new algorithm implements tuning rules that work on the network structure and on the hyper-parameter' search space to drive the training of the network towards better results. Some experiments were performed over standard datasets to demonstrate the improvement provided by DNN-Tuner. Future works will be focused on the implementation of the tuning rules with symbolic languages (either abductive as in [8], or probabilistic as in [1]) in order to achieve higher modularity, and interpretability (as well as explainability) for them.

Acknowledgements

This work was also supported by the National Group of Computing Science (GNCS-INDAM). The first author is supported by a scholarship funded by Emilia-Romagna region, under POR FSE 2014-2020 program.

References

1. Alberti, M., Cota, G., Riguzzi, F., Zese, R.: Probabilistic logical inference on the web. In: AI*IA 2016: Advances in Artificial Intelligence - XVth International Conference of the Italian Association for Artificial Intelligence, Genova, Italy, November 29 - December 1, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10037, pp. 351–363. Springer (2016)
2. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of Machine Learning Research **13**(Feb), 281–305 (2012)
3. 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)

4. Commons, W.: File:gpparbayesanimationsmall.gif — wikimedia commons, the free media repository (2019), `https://commons.wikimedia.org/w/index.php?title=File:GpParBayesAnimationSmall.gif&oldid=380025760`, online; accessed 16-January-2020
5. Dewancker, I., McCourt, M., Clark, S.: Bayesian optimization primer (2015)
6. Dongarra, J., Eijkhout, V.: Self-adapting numerical software and automatic tuning of heuristics. In: International Conference on Computational Science. pp. 759–767. Springer (2003)
7. Frazier, P.I.: A tutorial on bayesian optimization. arXiv preprint arXiv:1807.02811 (2018)
8. Gavanelli, M., Lamma, E., Riguzzi, F., Bellodi, E., Zese, R., Cota, G.: An abductive framework for datalog± ontologies. In: Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015. CEUR Workshop Proceedings, vol. 1459, pp. 128–143. CEUR-WS.org (2015)
9. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. CoRR **abs/1502.03167** (2015), `http://arxiv.org/abs/1502.03167`
10. Jalali, A., Azimi, J., Fern, X.Z.: Exploration vs exploitation in bayesian optimization. CoRR **abs/1204.0047** (2012), `http://arxiv.org/abs/1204.0047`
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. Korichi, Guillemot, M., Heusèle, C., Rodolphe: Tuning neural network hyperparameters through bayesian optimization and application to cosmetic formulation data. In: ORASIS 2019 (2019)
13. van Laarhoven, T.: L2 regularization versus batch and weight normalization. CoRR **abs/1706.05350** (2017), `http://arxiv.org/abs/1706.05350`
14. Lam, R., Willcox, K., Wolpert, D.H.: Bayesian optimization with a finite budget: An approximate dynamic programming approach. In: Advances in Neural Information Processing Systems. pp. 883–891 (2016)
15. Montavon, G., Orr, G., Müller, K.R.: Neural networks: tricks of the trade, vol. 7700. springer (2012)
16. Rai, A., Antonova, R., Song, S., Martin, W., Geyer, H., Atkeson, C.: Bayesian optimization using domain knowledge on the ATRIAS biped. In: 2018 IEEE International Conference on Robotics and Automation (ICRA). pp. 1771–1778. IEEE (2018)
17. Rasmussen, C.E.: Gaussian processes in machine learning. In: Summer School on Machine Learning. pp. 63–71. Springer (2003)
18. 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)