Online hyper-parameters optimization

Stefano Fioravanti, matr. 1806588
Master Degree in ”Artificial Intelligence and Robotics”
Sapienza Università di Roma
fioravanti.1806588@studenti.uniroma1.it

Abstract—**Nowdays a Neural Network can be designed in many ways, and find the best configuration is becoming a hard task. The configuration is a set of hyper-parameters, that describes the structures of the NN.**

Usually the optimal configuration is found training the model with different configurations sampled from the configuration space and then select the best one wrt the test performance.

This method implies that every configuration is trained and tested and that the chosen one represents only the best of the proposed configurations and not the actual optimal.

This paper studies an algorithm [1] to tunes the hyper-parameters from a given configuration, exploring the configuration space by itself and returning a better (and in the optimal region) configuration after the first training.

I. Introduction

Any ML model is composed by two sets of parameters:

- **elementary parameters** θ: the collection of weights and biases that interacts with the input signal, they are float numbers;
- **hyper-parameters (HPs)** Λ: the model's design features, they can be:
  - *continuous* (float numbers), usually used in layers with probability as additive noise and dropout;
  - *discrete* (integer numbers), used to describe quantities and dimensions, and to represent ordered string values;
  - *categorical* (string), represent a collaction of unordered string values such as collection of function or layer to implement.

A common assumption is that the hyper-parameters affect the training trajectory, so they have to be fixed during the whole training.

An algorithm that mixes the Bayesian Optimization and the HyperBand (BOHB) [2] tunes the system resources reserved for each execution (e.g. RAM, training epochs, early-stop's threshold) propotionally to its expected utility. In this way the algorithm focus on the most useful configurations (found by the Bayesian optimizer).

A more general approach has been proposed with Auto-WEKA 2.0 [3], which finds not only the optimal configuration, but also the optimal algorithm (from an updated list of popular learners). To solve the *combined algorithm selectionand hyperparameter optimization (CASH)* problem, WEKA includes the algoeithm in the configuration set. The Bayesian optimization based on Gaussian processes performs wells for low-dimensional problems, for that reason WEKA applies a tree-based optimization and the random-forest-based SMAC [4].

Taking in account only the continuous HPs, if the largest updates occur at the beginning of the training and then they become small enough, both the parameters set can be tuned simultaneously. This is the base idea for the proposed algorithm [1].

II. Proposed method

The method treats the HPs similarly to the elementary parameters and updates them using Stochastic Gradient Descent.

The train set is split in two sub-sets:

- **training** set (T1) for the elementary parameters' updates;
- **validation** set (T2) for the HPs' updates.

This paper focuses on the tuning of:

- L2 regularization (λ);
- input gaussian noise (n_0);

The HPs gradients' computation depends on how they are implemented.

A. L2 regularization

Regularization is a process of introducing an additional cost proportional to the squared weights, thus the cost function becomes

$$\tilde{C}_1(\theta,\lambda) = C_1(\theta) + \Omega(\theta,\lambda) \tag{1}$$

$$\Omega(\theta,\lambda) = \sum\nolimits_j \lambda_j \frac{\theta_j^2}{2} \tag{2}$$

The gradient of the validation cost wrt the λ is:

$$\nabla_\lambda C_2 = \nabla_\theta C_2 \cdot \nabla_\lambda \partial\theta = \nabla_\theta C_2(-\eta_1 \nabla_\lambda \nabla_\theta \tilde{C}_1) \tag{3}$$

Since the $\tilde{C}_1$ formula is known, it's possible to compute the double gradient, otherwise its computation would be computationally expensive.

The final gradient is:

$$\begin{aligned} \frac{\partial^2 \tilde{C}_1}{\partial\lambda_i \partial\theta_j} &= \frac{\partial^2}{\partial\lambda_i \partial\theta_j}\left(C_1(\theta) + \sum_k \lambda_k \lambda_k \frac{\theta_k^2}{2}\right) = \\ &= \frac{\partial}{\partial\theta_j}\frac{\theta_i^2}{2} = \theta_i \Rightarrow \nabla_\lambda \nabla_\theta \tilde{C}_1 = \theta \end{aligned} \tag{4}$$

$$\nabla_\lambda C_2 = \nabla_\theta C_2(-\eta_1 \nabla_\lambda \nabla_\theta \tilde{C}_1) = -\eta_1 \nabla_\theta C_2 \cdot \theta \tag{5}$$

B. Gaussian noise

The gaussian noise is applied directly during the forward propagation (the L2 regularization is added only to the final cost), so the gradient can be computed with the back propagation as usual with the elementary parameters.

Since the HP is the Gaussian's standard deviation, the gradient has to be multiplied by the noise used during the forward propagation.

Moreover, the gradient is usually computed through native methods, that required the parameter wrt which compute it.

A solution can be to compute the gradient with the native methods until the level summed with the Gaussian noise, and then multiply it by the noise.

In case the parameters in shared among the network, the update is the mean of the single components. The input noise's gradient is:

$$\nabla_{n_0} C_2 = \mathcal{N}(0, n_0) w_0 \nabla_{w_0} C_2 \tag{6}$$

III. Experiments

The project has been tested on the MNIST detaset, over a MLP network with 3 possible sizes:

- shallow $(1000 \times 1000 \times 1000)$;
- medium $(2000 \times 1000 \times 1000 \times 500)$;
- deep $(4000 \times 2000 \times 1000 \times 500 \times 250)$.

The used activation function is ReLu. The parameter sets have separate Adam optimizers with different learning rates: $\eta_1 = 10^{-3}$ for T1 and $\eta_2 = 10^{-3}$ for T2.

The intervals for the chosen HPs are:

- L2 regularization: $[-2.5, -5.5]$, expressed as $\log_{10}$, so the applied value will be 10^λ;
- Gaussian noise: $[0.0, 0.8]$.

IV. Results

The algorithm has been run 100 times with different configurations and sizes. The executions have been analized under three aspects: the efficiency, the performances and the time consumption.

A. Efficiency

The main algorithm's goal is to optimize the continuous hyper-parameters during the the training phase.

As showed in Fig.1, the HPs fall always in the same optimal region. Moreover, at every model size corresponds a different optimal region, this behavior highlights the difficulty to know a priori the best configuration.

The algorithm doesn't guarantee the convergence, so it works only if the initial configuration is enough close to the optimal region. The shallow models allow a greater interval of possible initial values, since they represent a simpler problem for the T1-T2 tuning.

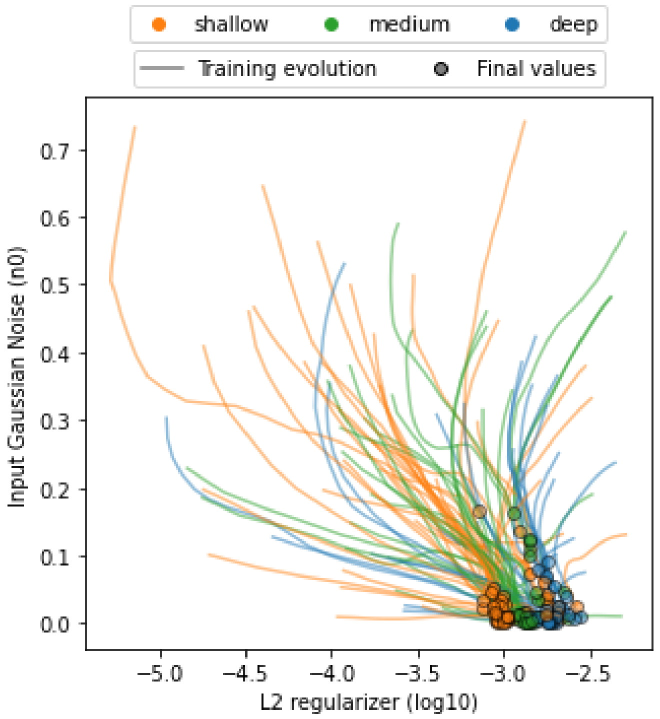

Fig. 1. HPs trends over training with final values highlighted.

In case of initial configurations too far from the optimal region, T1-T2 optimization may converge partially or nothign at all.

B. *Performances*

Find the optimal HPs' value means achieve a better performance. Fig.2 swohs the improvement given by T1-T2 optimization. Each dot represents a model trained with a fixed configuration: on the x-axis is the initial one, on the y-axis is the optimized one found by T1-T2.

The optimized models achieve $\sim 3\%$ of test error, while the initial ones vary a lot their performaces.

20% of the executions run with the initial configuration don't achieve an acceptable test error, due to not optimal HPs' values, but they achieve good performances after T1-T2 tuning.

58% of the samples lies next to the main diagonal, where the performances are similar for both the runs.

Fig.3 shows the performances at the end of T1-T2 tuning compared to the performances obtained running the found optimized configuration.

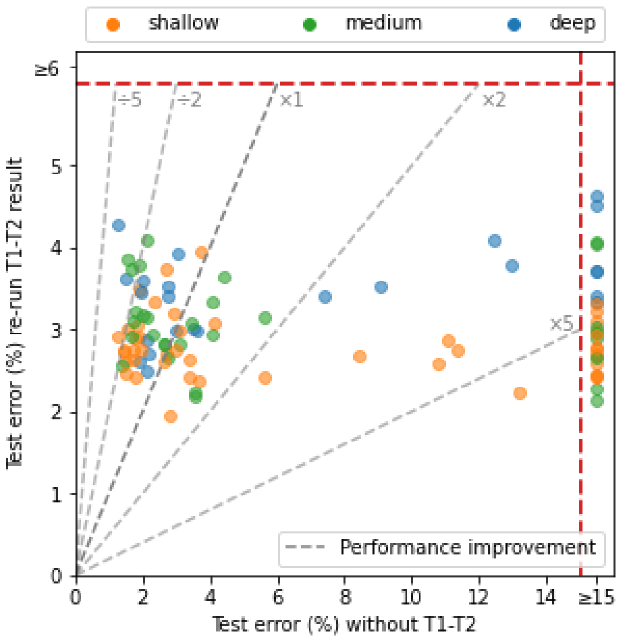

Fig. 2. Comparison of test errors when training with fixed hyperparameters before and after tuning them with T1-T2.

All the samples lies next to the main diagonal, thus it's not strictly required to run the optimized configuration, once T1-T2 found it.

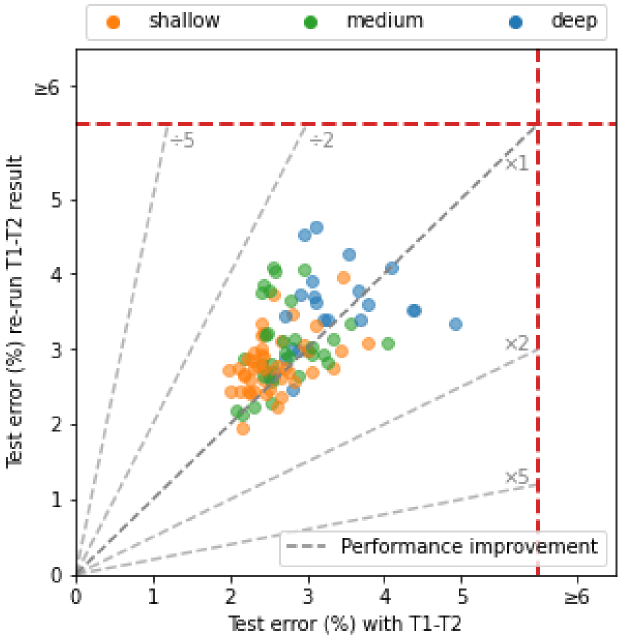

Fig. 3. Comparison of test errors at the end of the T1-T2 tuning and after the training with tuned configuration.

Of course, running the optimized configuration since the beginning of the training allows to achieve better results, for that reason it's possible to notice that the sample are slighty shifted on the upper side of the diagonal.

C. Time consumption

The average training time per epoch without T1-T2 is $15.14s$ (runs with initial and optimized configuration), with T1-T2 tuning it's $16.99s$.

Therefore T1-T2 algorithm required $1.85s$ per epoch, equal to an additional time cost of 12.22% to tune the input Gaussian noise and, more time consuming, the shared L2 regularier.

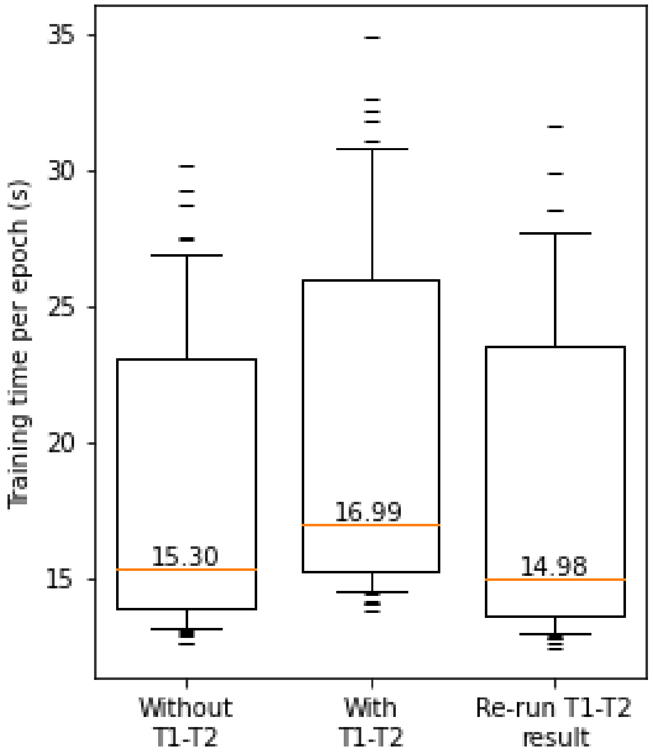

Fig. 4. Training time comparison

V. Conclusion & future works

The algorithm works well on continuous HPs, even if, in this way, many design features are cut out from the optimization.

To apply the algorithm it's necessary to compute the gradient for the backpropagation, this step makes the algorithm not easy to generalize and apply on different model. Once the gradient has been computed the algorithm can be easily applied, and it has a small impact on computation and training time.

A possible improvement is to merge this algorithm with other ones that take in account discrete and categorical HPs.

References

[1] J. Luketina, M. Berglund, K. Greff, and T. Raiko, "Scalable Gradient-Based Tuning of Continuous Regularization Hyperparameters," p. 9.

[2] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and Efficient Hyperparameter Optimization at Scale," *arXiv:1807.01774 [cs, stat]*, July 2018. http://arxiv.org/abs/1807.01774.

[3] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA," *The Journal of Machine Learning Research*, vol. 18, pp. 826–830, Jan. 2017.

[4] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential Model-Based Optimization for General Algorithm Configuration (extended version)," p. 24.