14th World Congress on Computational Mechanics (WCCM)
ECCOMAS Congress 2020)
Virtual Congress: 11-–15 January 2021
F. Chinesta, R. Abgrall, O. Allix and M. Kaliske (Eds)

ACCURACY OF NEURAL NETWORKS FOR SURROGATE MODELLING OF THE SAG OF HANGING CABLES

MATTHIEU ANCELLIN[1] **AND VINCENT LAURENT**[2]

[1] Université Paris-Saclay, ENS Paris-Saclay, CNRS, Centre Borelli, Gif-sur-Yvette, France
matthieu.ancellin@ens-paris-saclay.fr

[2] Eurobios SCB, Cachan, France
vlaurent@eurobios.com

Key words: Machine Learning, Neural Network, Solid Mechanics

Abstract. The creation of surrogate models is a classical problem in Machine Learning. The present paper is a case study of training a surrogate model for a real-life engineering problem: the computation of the sag of a cable hanging between two pylons. Neural networks have been trained using samples of the solution for several physical parameters. A parametric study of the role of three hyperparameters (the number of training samples, the size of the network and the initialization of gradient descent) is presented.

1 INTRODUCTION

In the last years, a lot of progress has been done in the field of Machine Learning (ML), and in particular to provide efficient and user-friendly implementations. ML tools are now easily accessible to experts from other fields, such as physicists and engineers. One of the applications of ML to physics and engineering is the construction of surrogate models, for instance when a simulation is computationally expensive to run.

Surrogate modelling Let us consider a real-valued function in dimension d, $f : [0,1]^d \to \mathbb{R}$, that is accessible but expensive to compute. The goal is to build an function $\tilde{f}$ that approximates f as accurately as possible, that is it minimizes the following cost function J:

$$J(\tilde{f}) = \frac{\|\tilde{f} - f\|_{L^2}}{\|f\|_{L^2}} = \frac{\left(\int_{[0,1]^d} |\tilde{f}(x) - f(x)|^2\right)^{\frac{1}{2}}}{\left(\int_{[0,1]^d} |f(x)|^2\right)^{\frac{1}{2}}}. \tag{1}$$

The integral of this L^2 norm is approximated by the normalized Root Mean Squared Error (RMSE) using N samples $(x_i, f(x_i))$ computed from uniformly distributed values $x_i \in [0,1]^d$:

$$J_N(\tilde{f}) = \frac{\left(\sum_{i=1}^N \frac{1}{N} |\tilde{f}(x_i) - f(x_i)|^2\right)^{\frac{1}{2}}}{\left(\sum_{i=1}^N \frac{1}{N} |f(x_i)|^2\right)^{\frac{1}{2}}}. \tag{2}$$

The set of N samples $(x_i, f(x_i))$ is also called *training dataset*.

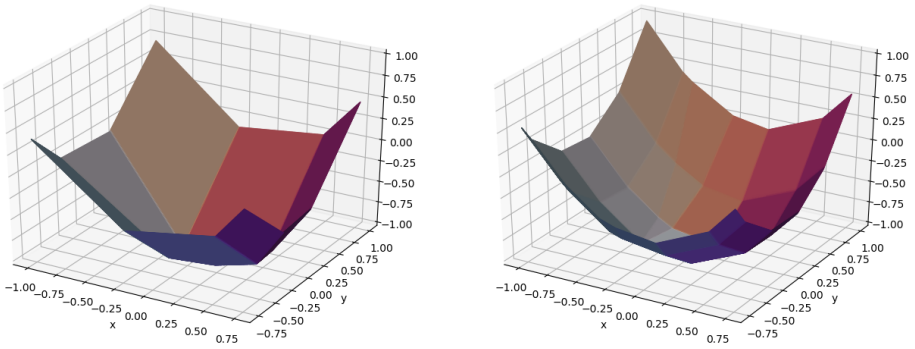

Figure 1: Approximations of the function $(x,y) \mapsto x^2 + y^2 - 1$ by a network of 4 neurons (*left*) and a network of 12 neurons (*right*). The color describes the direction of the gradient locally, in order to visualize the piecewise affine sections. The results have been manually picked among the local minima returned by the optimizer.

Neural networks In this paper, the surrogate models are neural networks. A feed-forward neural network [4] is a function $\mathbb{R}^d \mapsto \mathbb{R}^n$, built as the combination of simple elementary functions, called neurons, loosely inspired by the biological neurons. Each neuron is the composition of an affine function and a non-linear function, called activation function and denoted here σ. Neurons are combined in a network structure where each layer of neurons takes as inputs the outputs of the previous layer of neurons. The coefficients of the affine functions are parameters of the neural network and they will be fitted to the data by the learning process.

Neural networks can be seen as the generalization of the linear models of the following form (which includes polynomials or Finite Element decompositions):

$$\tilde{f}(x) = \sum_{i=1}^{n_1} a_i \phi_i(x), \tag{3}$$

where a_i are unknown weights, and ϕ_i are given basis functions. In a neural network, the basis functions ϕ_i are not fixed, but are parametrized by the introduction of more unknown weights. Indeed, a neural network with a single hidden layer can be written as

$$\tilde{f}(x) = \sum_{i=1}^{n_1} a_i \sigma \left(\sum_{j=1}^{n_2} B_{ij} \psi_j(x) \right). \tag{4}$$

The new basis functions ψ can be chosen as $\psi_i(x) = x_i$, or they can be decomposed in the same way as ϕ leading to a deeper network.

In this work, the Rectified Linear Unit (ReLU) activation function $\sigma(x) = \max(0,x)$ is considered. It is one of the simplest and the most popular activation functions used in the literature. With such a piecewise linear activation function, the neural network is a piecewise affine function, each neuron being responsible for one change of slope (see Figure 1). It can be intuitively understood that any function can be approximated with an arbitrary precision by piecewise affine functions. This statement can be proved mathematically and neural network are often characterized as "universal function approximators"[2, 9].

Finding a good approximation in a parameter space that can be of high dimension is usually done using a gradient descent algorithm. One of the strengths of the neural networks is the efficient implementations

that are available to compute their gradients (*back-propagation*). The main drawback of gradient descent is the risk of finding only a local minimum of the optimization problem. When many local minima exist, finding the global minimum (or just a decent local minimum) may require many trials and errors.

Objectives Even for a perfect optimization of J_N, the model would remain inaccurate with respect to the cost function J. Since increasing N by computing new samples can be costly, we look for the minimal number of samples necessary to obtain a given accuracy. Besides the fidelity of $\tilde{f}$ to f, another objective is the simplicity of the model $\tilde{f}$ which should be as fast as possible to evaluate. The goal of the present work is to quantify the trade-off between these contradictory objectives.

The usual practice in ML is to solve two nested optimization problems: finding the best coefficients for a given kind of model and finding the best kind of model. While the first one is automated with efficient optimization algorithms, the second one often relies on the experience of the user and many trials and errors. A goal of this paper is to provide some insights on this second optimization problem. The full automation of this second optimization is a current research topic in the ML community [5].

This kind of surrogate modelling differs slightly from classical statistical learning in two ways. Firstly, the evaluation of the function f is supposed to be purely deterministic and the goal of the surrogate model is to reproduce its result as accurately as possible. Unlike usual ML problem, we assume no noise on the measured samples $f(x_i)$. Secondly, we can choose the training data that we want to use. The choice of a sampling method in not discussed in details in this paper, but the number of samples is one of our main parameters.

Summary In Section 2, the physical problem of the sag of hanging cable is introduced and the methodology of the main study of this paper is discussed. In Section 3, the results of the study of the influence on some hyper-parameters of a neural network model (number of neurons, deepness, initialization of the coefficients) are presented.

2 METHODOLOGY OF THE STUDY

To evaluate the performance of neural networks in real conditions, an actual engineering problem has been considered.

Physical model The problem is a 1D Finite Element simulation of the motion and deformation of an high-voltage line between two support towers (see Figure 2). The simulation depends on 8 physical magnitudes: m is the mass per length unit of the cable (in kg/m), d is its diameter (in m), E_A is its axial stiffness (in N), H is the tension of the cable (in N), L_p is the span length between the two towers (in m), h is the altitude difference between the two towers (in m), U is the speed of the wind relatively to the cable (in m/s) and finally C_D is the drag coefficient of the wind. The function f that we want to model maps these eight parameters to the sag of the cable. For each of the input parameters, physical bounds have been provided.

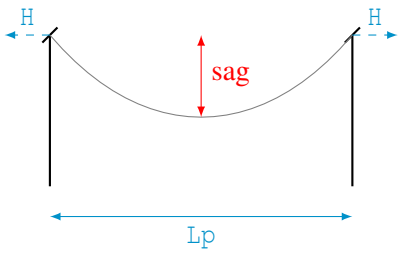

Figure 2: Schematics of the problem and some of its parameters.

The domain of definition of the function that we want to approximate is

then an 8-dimensional hyper-rectangle. The input domain has been scaled down to the $[0,1]^8$ hypercube using a linear scaling method. Sampling points (x_i) have been chosen using the Latin hypercube sampling method [7]. This method allows to cover uniformly the range of values of each of the input variables. Even if one of the parameters turns out to have no effect on the output, no redundant PDE problem is solved, as it would be with a regular grid sampling method.

Inputs The three hyper-parameters discussed in this paper are the following:

- the training size, that is N in (2). Training datasets have been precomputed, so for a given training size, the same dataset is used every time. (The random variability due to the sampling of the dataset is not discussed in this paper.) Values between 10 and 5000 samples are considered.
- the sizes of the layers of the neural network. Most of the following tests considered a single hidden layer, so this parameter is n_1 from (4). Networks containing between 1 and several thousands of neurons are considered.
- the random state for the initialization of the neural network, that is the seed of the random number generator used to chose an initial guess for the optimization algorithm. For each training size and network size, 1000 different initial states for the neural networks are randomly chosen using Scikit-learn's default initialization method (Glorot uniform sampling [3]). This parameter can be seen as a random variable and the results of the parametric study is presented as probability distributions of functions depending on this variable.

All computations have been done with the Scikit-learn open source Python library [6, 8]. The LBFGS optimizer have been used for all the following numerical results. Other settings are the defaults of Scikit-learn as of version 0.20.3.

Output The main output of interest is the accuracy of the result, quantified using a testing dataset different from the training set, as it is usually done in Machine Learning. Indeed, evaluating the model only on the value of (2) that has been reached during the optimization process would miss the fact that we are actually trying to minimize (1). To better evaluate the error with respect to (1), another independent error of the form (2) is defined by the computation of another set of samples, called *test dataset*. In our parametric study, the same test dataset has been used for all cases. It is as large as the largest training dataset (5000 samples) and has been computed with an independent random sampling.

Variant model: log scaling As we have mentioned earlier in this paper, neural networks with the ReLU activation function are piecewise affine functions. However, many models in physics and engineering are written instead as power laws of the form $\frac{x^a y^b}{z^c}$. (In physics, it rarely makes sense to add values of different "dimensions", whereas multiplying them can make sense.) This kind of model is actually just a linear function in log scale $\exp(a\log(x) + b\log(y) - c\log(z))$, so a simple rescaling of the training data is sufficient to train a power law with neural networks.

In the Sections 3.3 and 3.4, we conduct the same study as in the Sections 3.1 and 3.2, except for the log-scaling of the input data. In other words, the model $\tilde{f}$ is replaced with $\exp \circ \tilde{f} \circ \log$. The same datasets are used and the cost function is still the same approximation (2) of (1).

Another motivation for this model variant is the expected shape of the function f: our sag problem is

a complex version of the catenary equation, and thus it is expected to involve the hyperbolic cosine function and to behave as an exponential function.

3 RESULTS AND DISCUSSION

3.1 Role of the training size

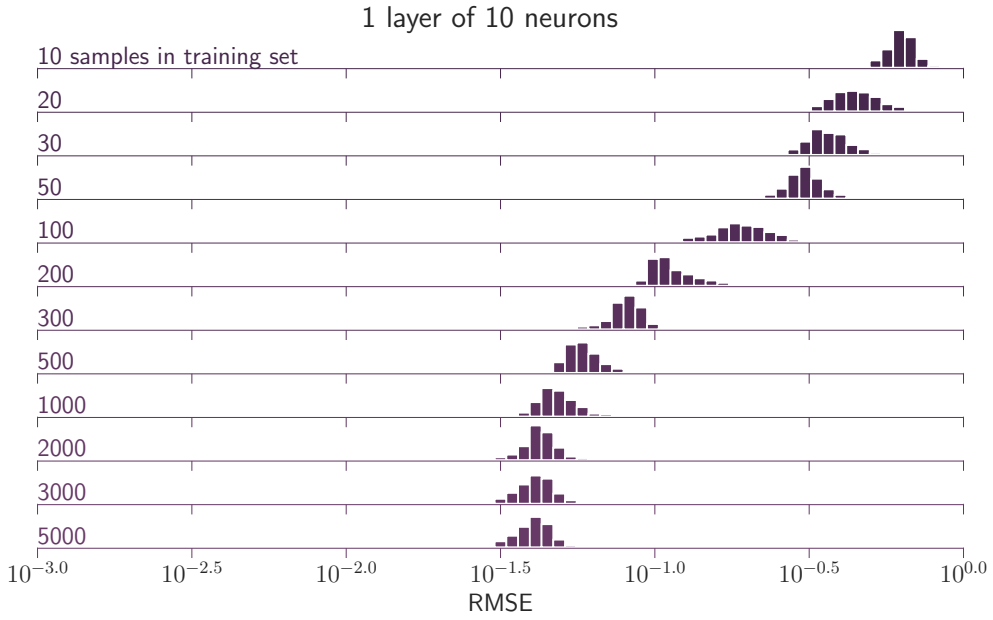

Figure 3: Probability distribution (with respect to the random initialization of the neural network) of the test RMSE in logarithmic scale for training datasets of different size. All trainings are done with a network of a single hidden layer of 10 neurons.

On Figure 3, the distribution of the test error has been plotted in log scale for different training datasets.

Firstly, we notice that for each case the distribution of test error has the shape of a narrow Gaussian distribution. All the local minima reached by the algorithm are very close from one another. It means that, in practice, the optimization method is not so sensitive to local minima and it might not be necessary to repeat the gradient descent so many times.

Secondly, we notice that the larger the dataset, the lower the error. Until a dataset of around 1000 samples, the error seems to follow a 1/2 convergence rate, that is RMSE $\propto 1/\sqrt{\text{training size}}$. This is similar to the convergence rate of the Monte-Carlo method, which is basically the approximation of (1) by (2). However, the accuracy stops improving above 2000 samples. A possible explanation is that all the degrees of freedom of the model are used to their best and the number of neurons is now the limiting parameter.

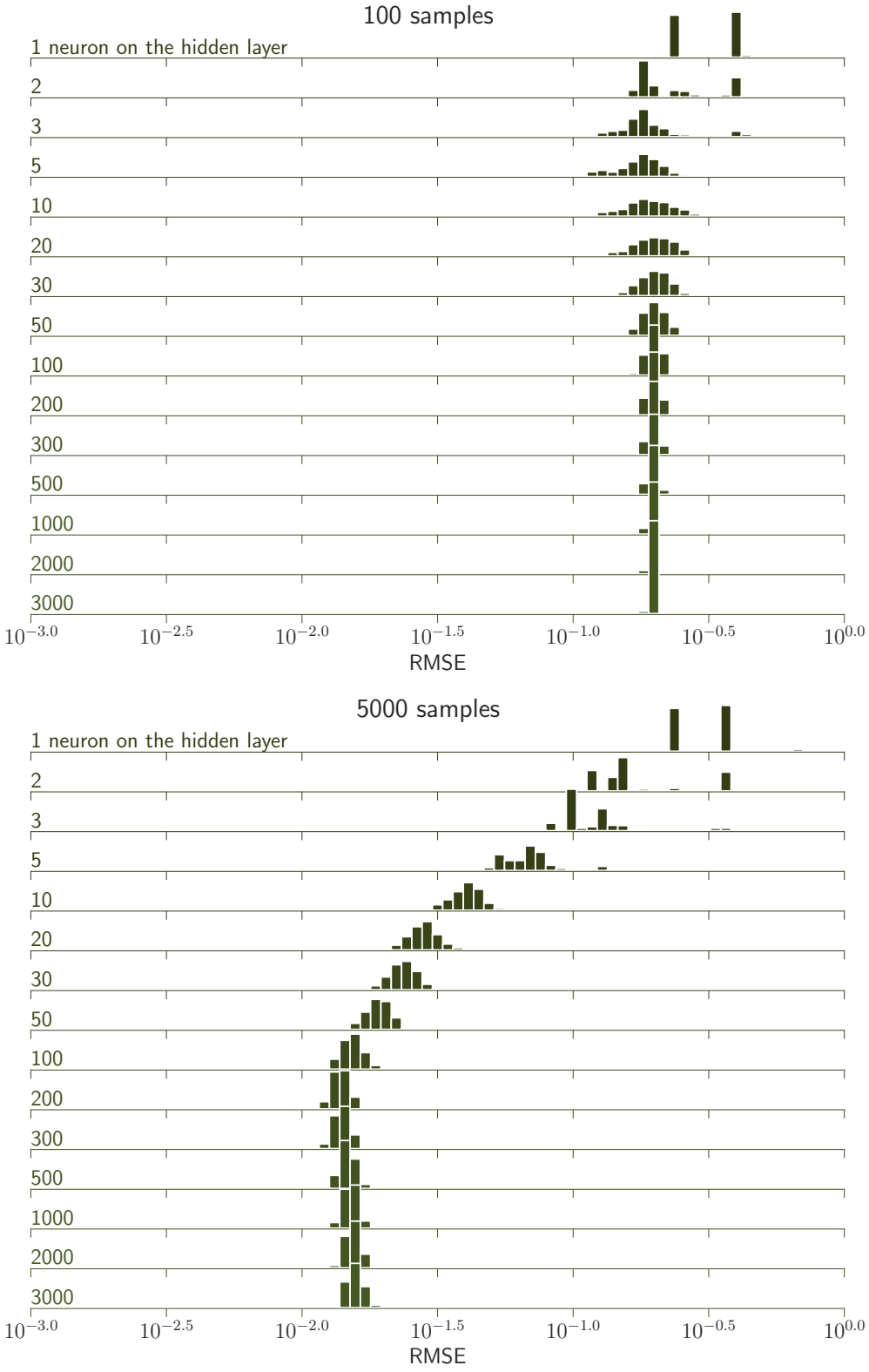

Figure 4: Probability distribution (with respect to the random initialization of the neural network) of the test RMSE in logarithmic scale for a network of one hidden layer with a different number of neurons. All trainings are done with a training dataset of 100 samples (top) or 5000 samples (bottom).

3.2 Role of the number of neurons

In the previous paragraph, we make the hypothesis of two domains in the space of hyperparameters: one where the accuracy is limited by the number of samples and one where the accuracy is limited by the number of neurons. To validate this hypothesis, the influence of the number of neurons have been plotted on Figure 4 for two training sets, assumed to be in the two domains.

On the top graph of Figure 4, the distribution of the errors has been plotted for several networks with a single hidden layer for 100 training samples. As expected, the number of neurons does not have a large influence on the averaged accuracy for this amount of samples.

However, the size of the network has clearly an influence on the variability of the error. For a small number of neurons, two local minima are visible. For the 1 neuron case, the results are shared equally between these two minima. For a higher number of neurons, the distribution takes the form of a Gaussian with a lower and lower standard deviation.

If we consider the lower value in the distribution of the errors, the best model are obtained when the number of neurons is around 5. A higher number of neurons leads to a slight loss of precision. It might be a weak sign of *overfitting*, that is a model with too many degrees of freedom in comparison with the number of samples.

Increasing the number of samples to 5000 on the bottom graph of Figure 4, the precision does not seem to be limited by the number of samples anymore and the number of neurons has a more important role. We now observe clearly the gain in precision from a single neuron until the model begins to overfit after 200 neurons. Between 1 and 100 neuron, the convergence rate is of order between 0.5 and 1.

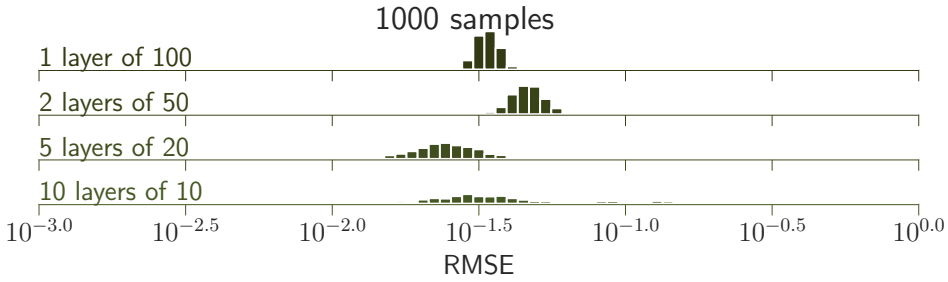

Figure 5: Probability distribution (with respect to the random initialization of the neural network) of the test RMSE in logarithmic scale for different shapes of neural networks with the same number of neurons but a different number of hidden layers. All trainings are done with a training dataset of 1000 samples.

On Figure 5, the distribution of the testing error has been plotted for several networks with the same number of neurons, but organized differently. The main observation is that a deeper network leads to a wider distribution of errors. The deep network is more versatile and better approximation can be found, but it is also harder to tune and the optimization process might more frequently get stuck in a bad local minimum.

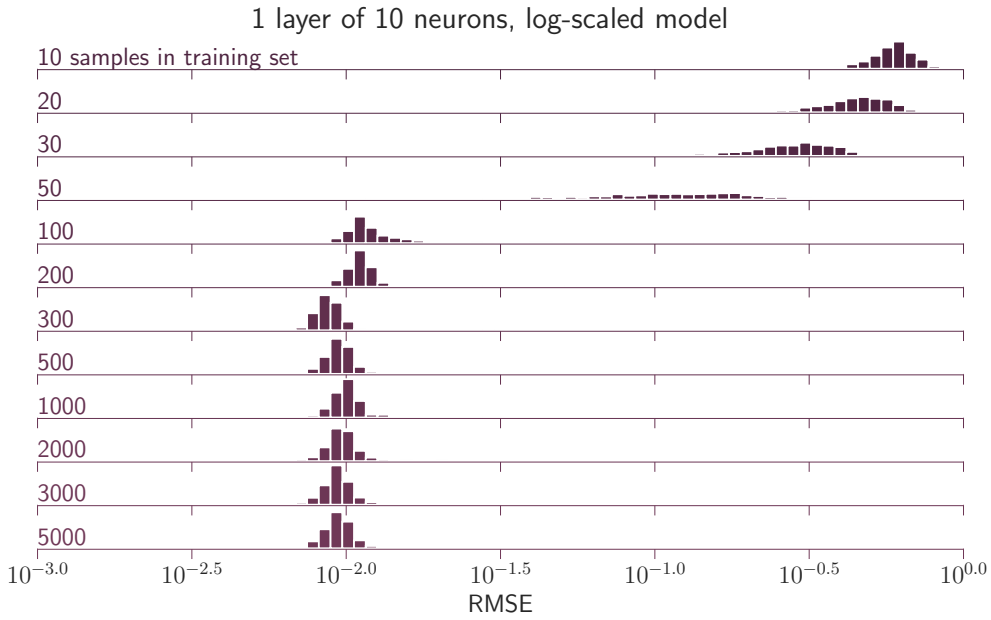

Figure 6: Probability distribution (with respect to the random initialization of the neural network) of the test RMSE in logarithmic scale for different training datasets of different size. All trainings are done with a network of a single hidden layer of 10 neurons.

3.3 Role of the training size for the log-scaled model

Figure 6 is the same as Figure 3 for a log-scaled model. All errors are now lower when using the log-scaled model. Fitting a power law model is a better match to the shape of the goal function f and thus interpolation between training points is more accurate. As in Figure 3, a plateau is reached in which an increase of the number of samples does not lead to an increase of the accuracy of the result. The plateau appears earlier and has a lower value for this model in log scale.

3.4 Role of the number of neurons for the log-scaled model

Figure 7 is the same as the top of Figure 4 for a log-scaled model. As in Section 3.3, all errors are lower when using the log-scaled model. As in Figure 4, two local mimina are visible for small networks and the worst local minima disappears when the number of neurons in the network increases.

The best error can be found for networks of around 5 neurons. The increase of the error that is observed when passing from 10 to 30 neurons can be interpreted as overfitting. For a higher number of neurons, the error slowly decreases. This phenomenon is the known *double descent phenomenon* [1]. The error due to overfitting can be understood as a random noise that can be mitigated by the averaging of many overfitting model. Thus a network with many more neurons than the overfitting state can behave like an ensemble average of overfitting networks, reducing the error.

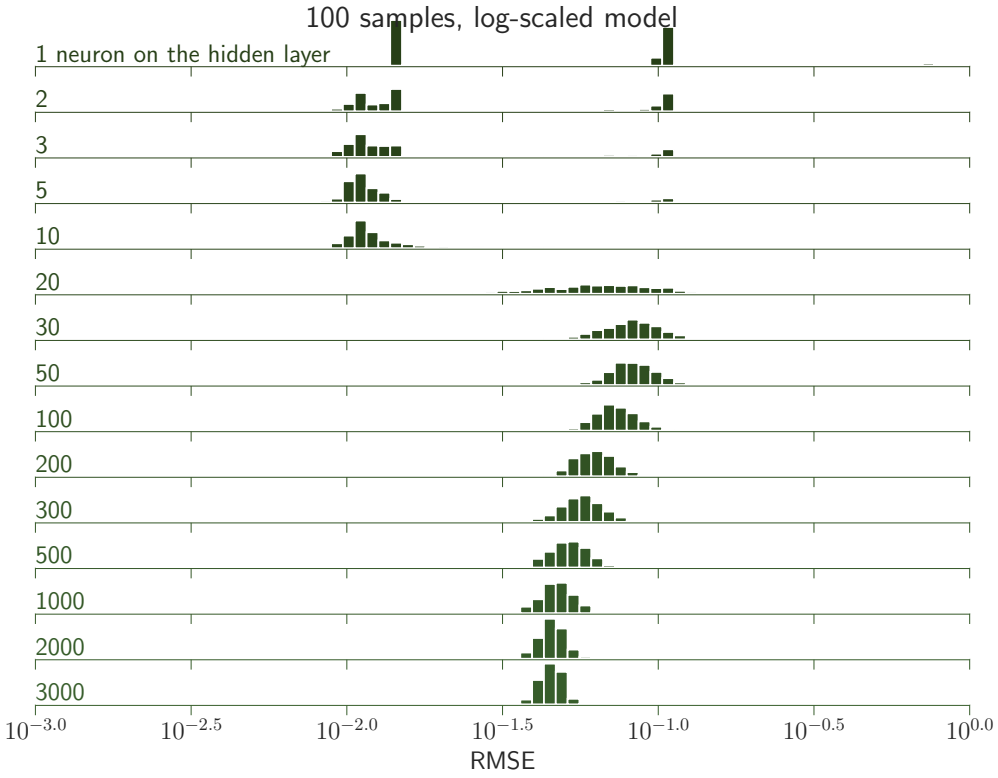

Figure 7: Probability distribution (with respect to the random initialization of the neural network) of the test RMSE in logarithmic scale for different networks with one hidden layer of a different number of neurons. All trainings have been done with a training dataset of 100 samples.

3.5 Discussion on the source of errors in neural networks

Figure 8 presents a summary of the influence of the training size and the number of neurons. For each configuration, the first decile of the error distribution has been displayed (instead of the minimal error which was noisier).

In the linearly scaled case (on the left), the error is mostly monotonous with respect to the two parameters of interest (except for some slight overfitting when the number of neurons gets higher). The optimal number of neurons to avoid overfitting is approximately one tenth of the number of samples.

In the logarithmically scaled case (on the right), the pattern is more complex. The function we are trying to model is close to linear in log scale and a single neuron gives a fair approximation. The effect of overfitting is much more visible, but the effect of the double descent is also observable for networks with a large number of neurons. Despite the model requiring less neurons than the linear-scaled model, no combination of parameters seems to be able to go much below a relative RMSE of 10^{-2}.

In statistical learning, the error of a model is often decomposed in two components: the bias and the variance. The bias is the error due to the lack of degrees of freedom in the model with respect to the complexity of the function to be modeled. The variance is the error due to overfitting, that is the use of too many degrees of freedom with respect to the number of samples.

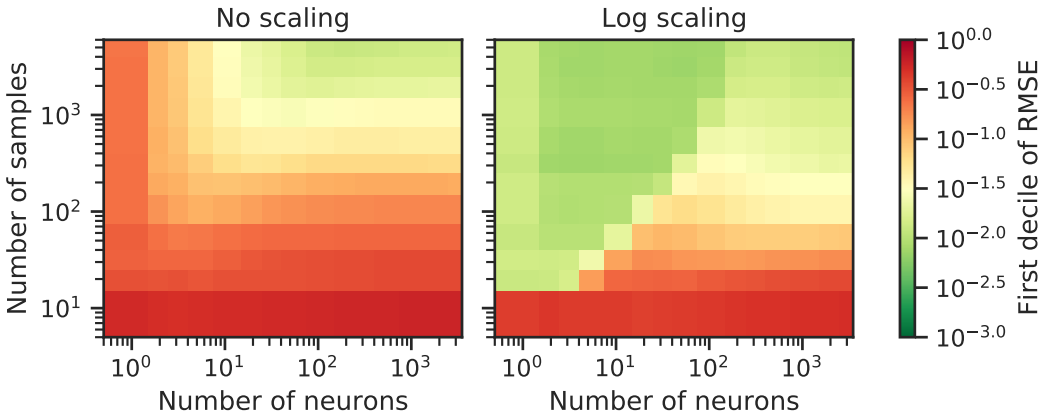

Figure 8: First decile of the probability distribution (with respect to the random initialization of the neural network) of the test RMSE for different sizes of training sets and a network of a single hidden layer with different number of neurons.

An interpretation of the previous results, and in particular Figure 8, in term of bias and sample is proposed in Figure 9. The space of hyperparameters is split in two domains. The boundary is approximately where the number of samples is 10 times the number of neurons.

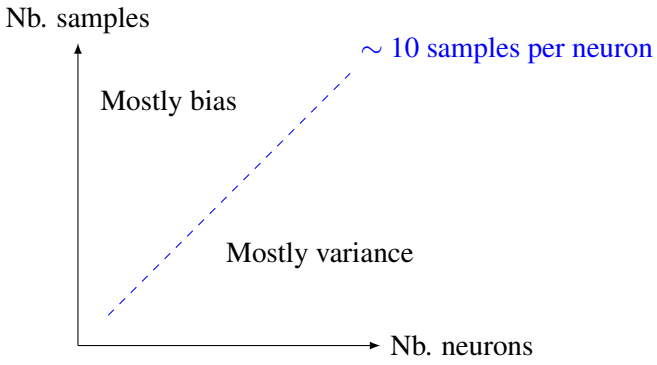

Figure 9: Interpretation of Figure 8 as two domains with different sources of error.

For cases above the line, the error is mostly due to bias. We observe as expected that the error is mostly independent of the number of samples and decreases with an increase of the number of neurons. The error due to bias is much smaller for the log-scaled model, as the form of the model fits better the modeled function.

For cases below the line, the error is mostly due to variance. As expected, variance increases then decreases with the number of neurons: it increases due to overfitting until the interpolation threshold, then decreases due to the central-limit-like behavior of double descent. Besides, variance is also sensitive to the number of samples: an increase in the number of samples reduces the overfitting [1]. Unlike bias, the log-scaling of the model does not have such a large effect on the amplitude of the variance error.

4 Conclusion

This paper is a case study of the influence of some hyperparameters on the accuracy of neural networks. Surrogate models for a real-life engineering problem have been computed by machine learning. The accuracy of the network is a non-trivial function of the hyperparameters. In particular, it is non-monotonous with respect to the number of neurons, and a careful parametric study is necessary to ensure

that the optimal surrogate model has been found.

In this study, the most efficient way to increase the accuracy was to use a log-scaling of the data to fit a power law instead of a linear model. The reason of the efficiency of this trick is unclear: it could be contingent to the relative simplicity of our problem (which is well approximated by an exponential); or it could be a clue of a deeper relevance of power laws to engineering problems.

Many other hyperparameters have been ignored in this study. In particular, the optimization algorithm used for the minimization of the cost function J_N has barely been mentioned. Some shallow tests showed that the LBFGS algorithm used in this study was up to one order of magnitude more accurate than the stochastic gradient descent algorithms implemented in Scikit-learn (SGD and Adam). The latter are known to be slower to converge, but are told to be more efficient to avoid local minima. Our results showed only a marginal effect of the existence local minima on the error, which comforted us in using the LBFGS method.

Our problem is relatively simple (e.g. it is monotonous with respect to each of its 8 parameters), so we considered using a simpler model, such as the linear Radial Basis Functions model (another example of model of the form of (3)):

$$\tilde{f}(x) = \sum_i a_i \phi(|x - x_i|) + b \quad \text{where } \phi(r) = \exp(-\gamma r^2). \tag{5}$$

The advantage of linear models is the efficient optimization methods available to fit the model. For instance, a Support Vector Machine (SVM) can optimize the number of terms in the model, avoiding a long parametric study such as the one of this paper. However, we could not reach the same accuracy with RBF as with neural networks. A possible explanation is that the RBF assume that x lives in an isotropic space, in which a variation of $1/\sqrt{\gamma}$ in every direction has the same consequence. This is obviously not the case with our engineering problem in which there is no arbitrary way to convert a variation of e.g. the diameter of the cable into a variation of its tension. Having a neural network with at least one hidden layer allows us to learn not only the coefficients a_i in the last layer, but also the anisotropy of the space of x in the previous one(s). This is not possible with a linear model.

Finally, let us focus attention on the fact that our physical problem was mostly considered as a black box. Indeed, the purpose of the paper is to discuss the possibilities of neural networks and not the physics of hanging cables. Actually, the best approach would have been to use as much physical knowledge as possible to guide the building of the surrogate model. This has been illustrated by the use of log-scaled data based on a physical argument, which greatly improved the quality of the surrogate model. In general, the crossing of physical models and machine learning models is still a promising open problem.

Acknowledgments

M.A. thanks CEA for support.

References

[1] Mikhail Belkin et al. “Reconciling modern machine-learning practice and the classical bias–variance trade-off”. In: *Proceedings of the National Academy of Sciences* 116.32 (2019), pp. 15849–15854. DOI: 10.1073/pnas.1903070116.

[2] George Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.
[3] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.
[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
[5] Xin He, Kaiyong Zhao, and Xiaowen Chu. “AutoML: A survey of the state-of-the-art”. In: *Knowledge-Based Systems* 212 (2021), p. 106622. DOI: 10.1016/j.knosys.2020.106622.
[6] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.
[7] Rickard Sjögren, Daniel Svensson, et al. *pyDOE2*. https://github.com/clicumu/pyDOE2.
[8] Jake VanderPlas. *Python data science handbook: Essential tools for working with data*. O’Reilly Media, Inc., 2016.
[9] Dmitry Yarotsky. “Error bounds for approximations with deep ReLU networks”. In: *Neural Networks* 94 (2017), pp. 103–114. DOI: 10.1016/j.neunet.2017.07.002.