Neural networks as an optimization tool for regression

Victor Azevedo Coscrato

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA INTERINSTITUCIONAL DE PÓS-GRADUAÇÃO EM ESTATÍSTICA UFSCar-USP

Victor Azevedo Coscrato

Redes neurais como método de otimização para regressão

Dissertação apresentada ao Departamento de Estatística – Des/UFSCar e ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Estatística - Programa Interinstitucional de Pós-Graduação em Estatística UFSCar-USP.

Orientador: Prof. Dr. Rafael Izbicki

São Carlos
Setembro de 2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA INTERINSTITUCIONAL DE PÓS-GRADUAÇÃO EM ESTATÍSTICA UFSCar-USP

Victor Azevedo Coscrato

Neural networks as an optimization tool for regression

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP and to the Departamento de Estatística – DEs-UFSCar, in partial fulfillment of the requirements for the degree of the Master joint Graduate Program in Statistics DEs-UFSCar/ICMC-USP.

Orientador: Prof. Dr. Rafael Izbicki

São Carlos
September 2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa Interinstitucional de Pós-Graduação em Estatística

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Victor Azevedo Coscrato, realizada em 02/09/2019:

Prof. Dr. Rafael Izbicki
UFSCar

Prof. Dr. Marcos Oliveira Prates
UFMG

Prof. Dr. Murilo Coelho Naldi
UFSCar

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Marcos Oliveira Prates e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

Prof. Dr. Rafael Izbicki

I dedicate this work to all who contributed to my journey,
especially my parents Marcos and Silmara and my companion Rafaela.

ACKNOWLEDGEMENTS

I am grateful to all the educators who have aroused my interest in science, especially my advisor Rafael Izbicki.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

ABSTRACT

COSCRATO, V. **Neural networks as an optimization tool for regression**. 2019. 62 p. Master dissertation (Master student joint Graduate Program in Statistics DEs-UFSCar/ICMC-USP) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Neural networks are a tool to solve prediction problems that have gained much prominence recently. In general, neural networks are used as a predictive method, that is, their are used to estimate a regression function. Instead, this work presents the use of neural networks as an optimization tool to combine existing regression estimators in order to obtain more accurate predictions and to fit local linear models more efficiently. Several tests were conducted to show the greater efficiency of these methods when compared to the usual ones.

Keywords: Regression, Neural networks, Ensembles, Local regression, Optimization.

RESUMO

COSCRATO, V. **Redes neurais como método de otimização para regressão**. 2019. 62 p. Master dissertation (Master student joint Graduate Program in Statistics DEs-UFSCar/ICMC-USP) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Redes neurais são uma ferramenta para resolver problemas de predição que ganharam muito destaque recentemente. Em geral, redes neurais são utilizados como um método preditivo, ou seja, estimando uma função de regressão. Este trabalho, no entanto, apresenta o uso de redes neurais como uma ferramenta de otimização para combinar estimadores de regressão já existentes de modo a obter predições mais precisas e ajustar modelos lineares locais de forma mais eficiente. Vários testes foram conduzidos para mostrar a maior eficiência desses métodos quando comparados aos usuais.

Palavras-chave: Regressão, Redes neurais, Combinação de regressões, Regressão local, Otimização.

LIST OF FIGURES

Figure 1 – Neural network example . . . 24
Figure 2 – Regressions comparison. While for some regions of x the linear fit outperforms the quadratic fit, in other regions the opposite happens. . . 30
Figure 3 – Meta-learners fits for Example 2.1.1. While Breiman's meta-learner is not able to fit the true regression satisfactorily, feature varying weights yield better fit. . . . 32
Figure 4 – Example of a UNNS neural network. . . . 32
Figure 5 – Example of the CNNS neural network. . . . 35
Figure 6 – Full NN-Stacking training process. . . . 36
Figure 7 – Weight distribution for the GPU kernel performance dataset. . . . 40
Figure 8 – Weight distribution for the music year dataset. . . . 41
Figure 9 – Weight distribution for the blog feedback dataset. . . . 43
Figure 10 – Weight distribution for the superconductivity dataset. . . . 44
Figure 11 – Example of a NLS neural network. . . . 47
Figure 12 – Top-left: MSE as function of the penalization strength λ. Notice that adding penalization decreases the NLS accuracy, as proposed by the trade-off. Top-right: Average squared gradient as function of λ. It can be seen that higher penalization values lead to smoother estimates for θ. Center and Bottom: True regression, NLS adjusts and fitted $\theta_1(x)$ for $\lambda = 0, 1$. While the non-penalized NLS yields better fit, the penalized version provides a smoother adjust. . . . 49
Figure 13 – Relevant and irrelevant features relationship with Y. Notice that the difference in the irrelevant features on different instances does not provide any information about Y. However, a typical kernel is affected in the same way by all features, relevant or irrelevant. . . . 52
Figure 14 – Top: Models predictive accuracy over different sample sizes for the Amazon fine foods dataset. Notice that NLS performance increases in comparison to the others as the sample size grows. Bottom: Models fitting time for different sample sizes. Notice that while bigger samples massively increase the fitting time for the LLS, the NLS suffers lower impact. Both methods are slow when compared to neural network regression and random forests. . . . 54
Figure 15 – MSE and average squared gradients for both training and test datasets. Higher penalization values leads to smoother $\Theta(\mathbf{x})$ estimates but higher train MSE. . . . 55

Figure 16 – Average extension error as function of the penalization strength. Notice that higher penalization values leads to more accurate prediction extensions. 56

LIST OF CHARTS

LIST OF ALGORITHMS

Algoritmo 1 – Back-propagation 25
Algoritmo 2 – UNNS 33
Algoritmo 3 – CNNS 35

LIST OF TABLES

Table 1 – Evaluation of model accuracy metrics for the GPU kernel performance dataset. . . . 40
Table 2 – Pearson correlation between base estimators prediction errors for the GPU kernel performance dataset. . . . 40
Table 3 – Evaluation of the model accuracy metrics for the music year dataset. . . 41
Table 4 – Pearson correlation between base estimators prediction errors for the music year dataset. . . . 41
Table 5 – Evaluation of model accuracy metrics for the blog feedback dataset. . . 42
Table 6 – Pearson correlation between base estimators prediction errors for the blog feedback dataset. . . . 42
Table 7 – Evaluation of model accuracy metrics for the superconductivity dataset. 43
Table 8 – Pearson correlation between base estimators prediction errors for the superconductivity dataset. . . . 44
Table 9 – Models mean scored errors across different irrelevant features number addition. While LLS is heavily affected by irrelevant features due to the issues they cause on sample weighting, NLS does suffer as much. . . . 51
Table 10 – MSE, MAE and their standard errors for the test set and the fitting times for each dataset. Notice that NLS and the neural network regression have similar predictive accuracy in 2 out of 4 datasets. NLS was more accurate in the Boston housing and the superconductivity data. Also, for every dataset, the results obtained by NLS are similar to the random forests and superior to LLS. The fitting time of NLS is high (especially on high dimensional data), being only overtaken by LLS in the bigger datasets. . . . 53

CONTENTS

1 INTRODUCTION 23
1.1 Neural networks 23
1.1.1 Loss function 24
1.1.2 Back-propagation 25
1.1.3 Further improvements 26
1.2 Neural networks as an optimization tool 26

2 NN-STACKING 29
2.1 Stacking regression estimators 29
2.2 Notation and Motivation 30
2.3 Methodology 31
2.3.1 Comparison with standard stacking methods 35
2.3.2 Selecting base regressors 37
2.3.3 Implementation details 37
2.4 Experiments 38
2.4.1 GPU kernel performance dataset 39
2.4.2 Music year dataset 39
2.4.3 Blog feedback dataset 42
2.4.4 Superconductivity dataset 42

3 NLS 45
3.1 The NLS 46
3.1.1 Implementation details 48
3.1.2 Connection to local linear estimators 50
3.2 Experiments 51
3.2.1 Sample size effect 53
3.2.2 NLS interpretation 54

4 FINAL 57
4.1 Relationship between the NN-Stacking and the NLS 57
4.2 Final Remarks and future work 57
4.2.1 NN-Stacking 57
4.2.2 NLS 58

BIBLIOGRAPHY 59

CHAPTER 1

INTRODUCTION

Neural networks are a tool to solve prediction problems that have gained much prominence recently. Generally, in a statistical framework, neural networks are used as a predictive method, that is, to directly estimate regression functions. However, neural networks can also be used as optimization tools for complex loss functions. This work shows the usage of neural networks as an optimization tool to combine existing regression estimators in order to (i) obtain more accurate predictions and (ii) fit local linear models more efficiently. Section 1.1 presents a brief review over neural networks, meanwhile Section 1.2 introduces the neural networks as an optimization method and presents some usage examples.

1.1 Neural networks

Artificial neural networks are a widespread tool in the machine learning community. Their ability to generate good predictive models (supervised learning) for various problems is demonstrated by several authors (DAYHOFF; DELEO, 2001; ZHANG; PATUWO; HU, 1998; KHAN *et al.*, 2001).

An artificial neural network is so named because its structural construction is a computational reproduction of the human nervous system, that is, an artificial neural network is a tangle of neurons connected to each other. These neurons are divided into different layers, being an input layer, an output layer, and intermediate or hidden layers.

The operation of an artificial neural network occurs by successive linear combinations of the features inserted in the input layer, which is formed by a neuron for each feature. Given input values x_i (input layer), each neuron within each layer (intermediate or output) calculates a weighting of the previous layer value and adds a scalar to it, that is, if $n_{i,j}$ is the j-th neuron in the i-th layer (i > 1), then $n_{i,j}$ calculates the expression

$$n_{i,j} = <\mathbf{h}_{i,j}, \mathbf{n}_{i-1}> + b_{i,j}, \tag{1.1.1}$$

where $\mathbf{n}_{i-1}$ is the vector of neurons of the preceding layer and $\mathbf{h}_{i,j}$ a vector of constants, or weights, associated with the neuron $n_{i,j}$, $<\mathbf{a},\mathbf{a}'> = \sum_{i=1}^{d} a_i a_i'$ the euclidean inner product and $b_{i,j}$ a bias parameter (intercept).

Hence, a complete layer $\mathbf{n_i}$ (i > 1) calculate the operation

$$\mathbf{n}_i = H_i \mathbf{n}_{i-1} + \mathbf{b}_i \tag{1.1.2}$$

where H_i is a matrix with columns $\mathbf{h}_{i,j}$ and $\mathbf{b}_i = [b_{i,j}]$ a column vector of bias.

There may also be, on each layer, an activation function, which is useful to determine a scale for the output value of the same. That is, say $\sigma_i(.)$ is the activation function of the i-th layer, then, this layer calculates,

$$\mathbf{n}_i = \sigma_i(H_i \mathbf{n}_{i-1} + \mathbf{b}_i) \tag{1.1.3}$$

The figure 1 schematically represents a neural network with 4 input variables, 2 output variables, 1 hidden layer with 5 neurons, and g activation function on the output layer neurons.

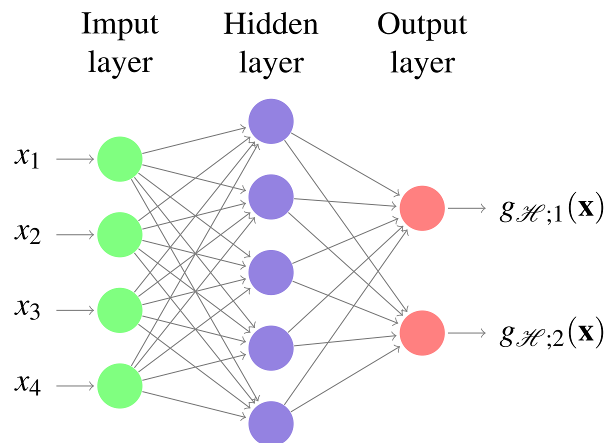

Figure 1 – Neural network example

The parameters related to a neural network are the set formed by the matrices $\mathbf{H}_i$ and the vectors $\mathbf{b}_i$ defined in the equation 1.1.2, denoted by $\mathscr{H}$.

1.1.1 Loss function

One core element of a neural network is the loss function. fixed an neural net architecture $f_{\mathscr{H}}$ with set of parameters $\mathscr{H}$, the loss (or cost) function for a given sample (x,y) evaluate what is the cost of using $f_{\mathscr{H}}$ to predict y using x. For instance, in predictive scenarios, the quadratic loss function is widely used, its defined by,

$$L(x,y;f_{\mathscr{H}}) = (y - f_{\mathscr{H}}(x))^2$$

The methods presented in both Chapters 2 and 3 are examples where choosing a suitable loss function and tweaking it enable a neural network to solve miscellaneous problems.

To fit a neural network, we assume that the loss function follows the assumption that the loss value for a given data set $D = (X, Y)$ must be written as a combination of the separated loss for each sample (x_i, y_i). In the most simple case,

$$L(X, Y; f_{\mathscr{H}}) = \frac{1}{n} \sum_{i=1}^{n} L(x_i, y_i; f_{\mathscr{H}}) \tag{1.1.4}$$

1.1.2 Back-propagation

To obtain optimal parameters $\mathscr{H}^*$, neural networks use the gradient descent method. In this method, given a loss function L, we obtain its derivatives with respect to each of the parameters and then slowly update the parameters to the opposite direction of the gradient. For instance, given a loss function $L(x_i, y_i, f_h)$, a single parameter h from a neural network f is updated as,

$$h \to h - \lambda \frac{\partial L(x_i, y_i, f_h)}{\partial h}$$

where λ is the learning rate hyper-parameter, that controls how slowly h is updated. Hence, estimating the gradient for given a data set $D = (X, Y)$ is the first step to optimize a neural network.

The back-propagation method, is an algorithm that uses the differentiation chain rule to obtain the gradient layer by layer trough the network. Rojas (1996) and Nielsen (2015) give expansive overviews and further discussions about the method, this section presents a brief review of the latter.

Algorithm 1 shows how the back-propagation calculate the derivatives for every parameter given an initial value $\mathscr{H}^0$ for a single sample (x_i, y_i).

Algorithm 1 – Back-propagation

Input: A training sample (x_i, y_i), an architecture f (hidden layers, its sizes and activation functions), initial parameters $\mathscr{H}^0$ and a loss function L. The symbol $\odot$ denotes the Hadamard product.
Output: The estimated loss gradient on $\mathscr{H}^0$.

1: **Feed-forward**: For each layer i = 2, 3, ..., I, compute $\mathbf{n}_i$,
2: **Output error**: Compute the loss $L(x_i, y_i, f_{\mathscr{H}^0})$ and the vector $\delta^I = \nabla_{n_I} L \odot \sigma_I'(n_I)$,
3: **Back-propagation**: For each i = I-1, I-2, ..., 2, compute $\delta^i = (H_i^t \delta^{i+1}) \odot \sigma_i'(n_i)$
4: **Output**: The gradient of L is $\frac{\partial L}{\partial H_i(j,k)} = n_{i-1}(k)\delta^i(j)$ and $\frac{\partial L}{\partial b_i(j)} = \delta^i(k)$

As long as the loss function respects equation 1.1.4, then, given a data set $D = (X, Y)$ with n samples, for a parameter h, it follows the relationship,

$$\frac{\partial L(x,y,f_h)}{\partial h} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L(x_i,y_i,f_h)}{\partial h} \tag{1.1.5}$$

Equation 1.1.5 provides a way to update parameters using a batch of data rather then a single sample. In practice, a data set is split into several mini-batches and this method is used to update the parameters using each of these mini-batches. For instance, a parameter h is update using a batch $(x_1,y_1),(x_2,y_2),...,(x_b,y_b)$ as,

$$h \to h - \frac{\lambda}{b}\sum_{i=1}^{b}\frac{\partial L(x_i,y_i,f_h)}{\partial h}$$

After using every mini-batch, these are resampled and another training epoch occurs until a stop criteria (for instance, a desired loss on a test set) is achieved.

1.1.3 Further improvements

As a neural network is a highly parameterized model, over-fitting is a common feature (TETKO; LIVINGSTONE; LUIK, 1995). Hence, some additional optimization tools might me used to prevent it.

Hinton *et al.* (2012) suggests the dropout method, which consists of substituting neuron values by 0 on the feed-forward phase of the back-propagation algorithm with some probability p. This method encourages the network to find new paths along its architecture, not relying too much on a specific group of neurons.

Batch normalization, suggested by Ioffe and Szegedy (2015) is another way of dealing with over-fitting, it suggests normalizing layers to have mean zero and unitary variance. This should at the same time reduce training speed and prevent over-fitting, as it works as a type of regularization method. This might also be used together with dropout.

Choosing a suitable initial value for the network parameters is also an improvement that increase training speed meanwhile guarantee that the gradient decent do not get stuck in a local minimum nor saddle points. Glorot and Bengio (2010) propose an initialization method.

Using other optimization algorithms rather then the gradient decent is also possible and reasonable in some cases. Some of these algorithms, such as the Adam optimizer (KINGMA; BA, 2014) differ from gradient decent by applying a variable learning rate.

1.2 Neural networks as an optimization tool

Neural networks have their more frequent usage in predictive scenarios, that is, when the goal is estimating the regression function $E[Y|x]$. There exists, although, various

cases where neural networks are used to solve miscellaneous optimization problems. The difference between both usages consists mostly replacing the usual output layer Y with another set of desired values.

On *reinforcement learning* (KAELBLING; LITTMAN; MOORE, 1996), for example, there is a limited set of possible actions that can be taken at a certain time t and some information about this time. A neural network is then used to estimate the probabilities of taking each action, in other words, a transition matrix. This neural network is designed to maximize efficiency of the taken actions, the concept of how good an taken action was is controlled by a suitable loss function.

Another example is the *word2vec* (GOLDBERG; LEVY, 2014) framework, which is a neural network interface to natural language representation. In this case we estimate the probability of each context to be associated to an input word using a neural network. The neural network is used to maximize the likelihood function for the pairs word-context in the training data. The most useful part of such technique is the hidden layer of the neural network, such layer values for a given word is a latent vector representation of the word.

Chapters 2 and 3 will propose two other methods that uses neural networks as an optimization tool, the NN-Stacking and the NLS. In both cases, the final objective is estimating a regression function with a predetermined shape; the neural networks are then used to optimize the parameters of models with these given shape.

CHAPTER

2

NN-STACKING

2.1 Stacking regression estimators

The standard procedures for model selection in prediction problems is cross-validation and data splitting. However, such an approach is known to be sub-optimal (DŽEROSKI; ŽENKO, 2004; DIETTERICH, 2000; SILL *et al.*, 2009). The reason is that one might achieve more accurate predictions by *combining* different regression estimators rather then by *selecting* the best one. Stacking methods (ZHOU, 2012) are a way of overcoming such a drawback from standard model selection.

A well known stacking method was introduced by Breiman (1996). This approach consists in taking a linear combination of base regression estimators. That is, the stacked regression has the shape $\sum_{i=1}^{k} \theta_i g_i(\mathbf{x})$, where g_i's are the individual regression estimators (such as random forests, linear regression or support vector regression), θ_i are weights that are estimated from data and $\mathbf{x}$ represents the features.

Even though this linear stacking method leads to combined estimators that are easy to interpret, it may be sub-optimal in cases where models have different *local accuracy*, i.e., situations where the performance of these estimators vary over the feature space. Example 2.1.1 illustrates this situation.

Example 2.1.1. Consider predicting Y based on a single feature, x, using the data in Figure 2. We fit two least squares estimators: $g_1(\mathbf{x}) = \theta_{01} + \theta_{11}x$ and $g_2(\mathbf{x}) = \theta_{02} + \theta_{12}x^2$. None of the models is uniformly better; for example, the linear fit has better performance when $x \leq -5$, but the quadratic fit yields better performance for $x \in (-2.5, 2.5)$. One may take this into account when creating the stacked estimator by assigning different weights for each regression according to x: while one can assign a larger weight to the linear fit on the regime $x \leq -5$, a lower weight should be assigned to it if $x \in (-2.5, 2.5)$.

It is well known that different regression methods may perform better on different regions of the feature space. For instance, because local estimators do not suffer from

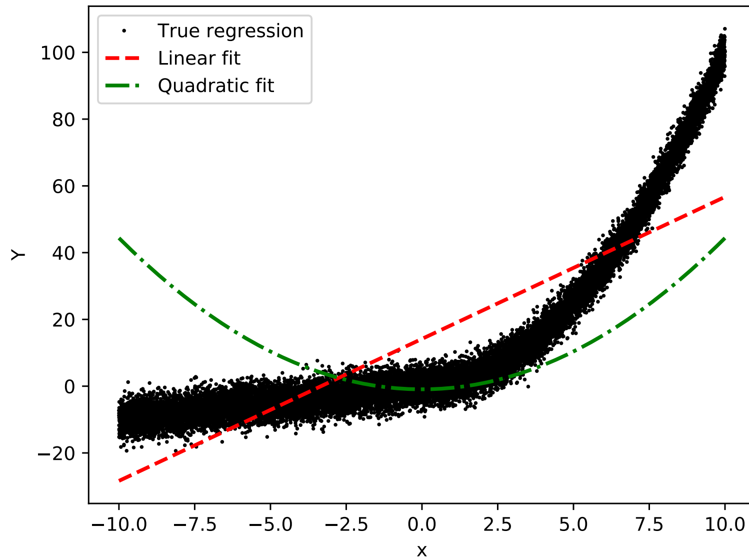

Figure 2 – Regressions comparison. While for some regions of x the linear fit outperforms the quadratic fit, in other regions the opposite happens.

boundary effects, they achieve good performance closer to the edges of the feature space (FAN; GIJBELS, 1992). Random forests, on the other hand, implicitly perform feature selection, and thus may have better performance in regions where some features are not relevant (BREIMAN, 2001).

In this work we improve Breiman's approach so that it can take local accuracy into account. That is, we develop a meta-learner that is able to learn which models have higher importance on each region of the feature space. We achieve this goal by allowing each parameter θ_i to vary as a function of the features $\mathbf{x}$. In this way, the meta-learner can adapt to each region of the feature space, which yields higher predictive power. Our approach keeps the local interpretability of the linear stacking model.

The remaining of the Chapter is organized as follows. Section 2.2 introduces the notation used in the paper, as well as our method. Section 2.3 shows details on its implementation. Section 2.4 shows applications of our method to a variety of datasets to evaluate its performance. Section 4.2.1 summarizes the main results and proposes further researches.

2.2 Notation and Motivation

The stacking method proposed by Breiman (1996) is a linear combination of k regression functions for a label $Y \in \mathbb{R}$. More precisely, let $g_{\mathbf{x}} = (g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots, g_k(\mathbf{x}))'$ be a vector of regression estimators, that is, $g_i(\mathbf{x})$ is an estimate of $\mathbf{E}[Y|\mathbf{x}], \forall i = 1, 2, \ldots, k$. The linear stacked regression is defined as

$$G_\theta(\mathbf{x}) := \sum_{i=1}^{k} \theta_i g_i(\mathbf{x}) = \theta' g_{\mathbf{x}} \tag{2.2.1}$$

where $\theta = (\theta_1, \theta_2, \dots, \theta_k)'$ are meta-parameters. One way to estimate the meta-parameters using data $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ is through the least squares method, computed using a leave-one-out setup:

$$\arg\min_{\theta} \sum_{i=1}^{n} (y_i - G_{\theta}^{(-i)}(\mathbf{x}_i))^2 = \arg\min_{\theta} \sum_{i=1}^{n} (y_i - \theta' g_{\mathbf{x}_i}^{(-i)})^2, \tag{2.2.2}$$

where $g_j^{(-i)}(\mathbf{x}_i)$ is the prediction for $\mathbf{x}_i$ made by the j-th regression fitted without the i-th instance. Note that it is important to use this hold-out approach because if the base regression functions $g_1(\mathbf{x}), g_2(\mathbf{x}), \dots, g_k(\mathbf{x})$ are constructed using the same data as $\theta_1, \dots, \theta_k$, this can cause $G_{\theta}(\mathbf{x})$ to over-fit the training data.

In order for the stacked estimator to be easier to interpret, Breiman (1996) also requires θ_i's to be weights, that is $\theta_i \geq 0 \, \forall i = 1, 2, \dots, k$ and that $\sum_{i=1}^{k} \theta_i = 1$.

Even though Breiman's solution works on a variety of settings, it does not take into account that each regression method may perform better in distinct regions of the feature space. In order to overcome this limitation, we propose the *Neural Network Stacking* (NNS) which generalizes Breiman's approach by allowing θ on Equation 2.2.1 to vary with $\mathbf{x}$. That is, our meta-learner has the shape

$$G_{\theta}(\mathbf{x}) := \sum_{i=1}^{k} \theta_i(\mathbf{x}) g_i(\mathbf{x}) = \theta_x' g_{\mathbf{x}}, \tag{2.2.3}$$

where $\theta_x := (\theta_1(\mathbf{x}), \theta_2(\mathbf{x}), \dots, \theta_k(\mathbf{x}))'$. In other words, the NNS is a *local linear* meta-learner. Example 2.2.1 shows that NNS can substantially decrease the prediction error of Breiman's approach.

Example 2.2.1. We fit both Breiman's linear meta-learner and our NNS local linear meta-learner to the models fitted in Example 2.1.1. Figure 3 shows that Breiman's meta-learner is not able to fit the true regression satisfactorily because both estimators have poor performance on specific regions of the data. On the other hand, feature-varying weights yield a better fit.

2.3 Methodology

Our goal is to find $\theta_{\mathbf{x}} = (\theta_1(\mathbf{x}), \dots, \theta_k(\mathbf{x}))'$, $\theta_i : \mathscr{X} \longrightarrow \mathbb{R}$, that minimizes the mean squared risk,

$$R(G_{\theta}) = \mathbf{E}\left[(Y - G_{\theta}(\mathbf{X}))^2\right],$$

where $G_{\theta}(\mathbf{x})$ is defined as in Equation in 2.2.3.

We estimate $\theta_{\mathbf{x}}$ via an artificial neural network. This network takes $\mathbf{x}$ as input and produces an output $\theta_{\mathbf{x}}$, which is then used to obtain $G_{\theta}(\mathbf{x})$. To estimate the weights of

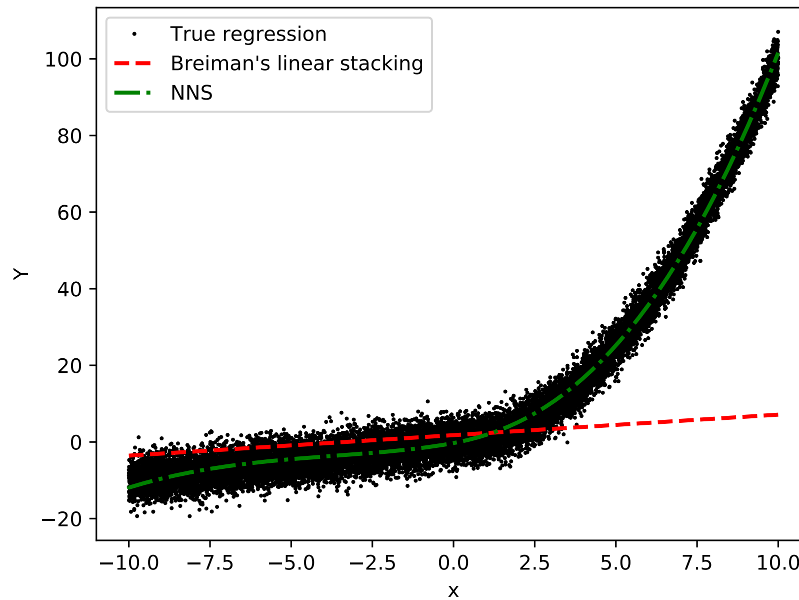

Figure 3 – Meta-learners fits for Example 2.1.1. While Breiman's meta-learner is not able to fit the true regression satisfactorily, feature varying weights yield better fit.

the networks, we introduce an appropriate loss function that captures the goal of having a small $R(G_\theta)$. This is done by using the loss function

$$\frac{1}{n}\sum_{k=1}^{n}(G_\theta(\mathbf{x}_k)-y_k)^2.$$

Notice that the base regression estimators are used only when evaluating the loss function; they are not the inputs of the network. With this approach, we allow each $\theta_i(\mathbf{x})$ to be a complex function of the data. We call this method *Unconstrained Neural Network Stacking* (UNNS). Figure 4 illustrates a UNNS that stacks 2 base estimators in a regression problem with four features.

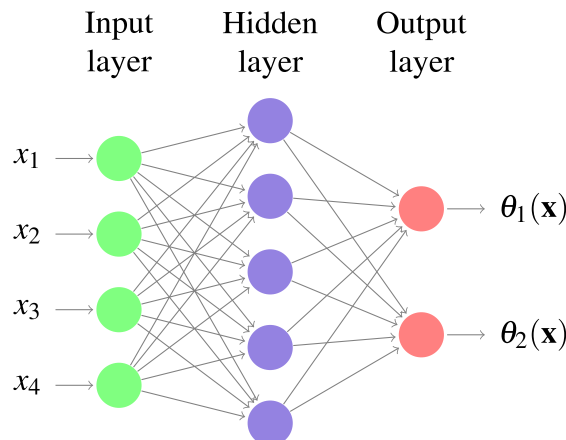

Figure 4 – Example of a UNNS neural network.

In addition to the linear stacking, this approach allows the user to easily take advantage of the neural network architecture by directly adding a network output node,

$\phi(\mathbf{x})$, to the stacking. That is, we also consider a variation of UNNS which takes the shape

$$G'_\theta(\mathbf{x}) = \theta'_\mathbf{x} g_\mathbf{x} + \phi(\mathbf{x}).$$

This has some similarity to adding a single neural network estimator to the stacking. However, we use the same architecture to create the additional term, mitigating computation time. Algorithm 2 shows how this method is implemented. In order to avoid over-fitting, θ_i's and g_i's are estimated using different folds of the training set.

Algorithm 2 – UNNS

Input: Estimation algorithms $g = (g_1, g_2, \dots, g_k)'$, a dataset $D = (X,Y)$ with n instances (rows), a neural network N, features to predict $X^{(p)}$, the amount of folds F.
Output: Predicted values $y^{(p)}$.

1: Let $I = \{I_o : o \in \{1,2,...,F\}\}$ be a random F-fold partition of the dataset instances, let $I^{(X)}$ refer to a partition of features and $I^{(Y)}$ refer to a partition of the response variable, both being partitioned on the same indices (i.e.: $I_o(i) = (I_o^{(X)}(i), I_o^{(Y)}(i))$ for every $o \in \{1,2,...,F\}$ and every $i \in \{1,2,...,n\}$), with the partition of indices $\{1,2,...,n\}$ represented by $I^{(l)}$.
2: Let P be a (n,k) matrix.
3: For $o \in \{1,2,...,F\}$ and $j \in \{1,2,...,k\}$, fit g_j to $D \backslash I_o$, then use the fitted model to predict $I_o^{(X)}$ and store these predicted values on P (in column j and lines corresponding to $I_o^{(l)}$)).
4: Let $\{g_1^{(f)}, g_2^{(f)}, ..., g_k^{(f)}\}$ be the models g fitted using the whole dataset D.
5: Train the neural network N with each input instance i given by a row of X; with $\theta(X_i) = (\theta_1(X_i), \theta_2(X_i), ..., \theta_k(X_i))$ and a scalar $\phi(X_i)$ as outputs; and with loss function given by $(\sum_{j=1}^k \theta_j(X_i)P_{ij} + \phi(X_i) - y_i)^2$ (note: the additional scalar ϕ is optional, i.e.: it can be set to zero).
6: For each instance i of $X^{(p)}$, the corresponding predicted value $Y_i^{(p)}$ is then given by $\sum_{j=1}^k \theta_j(X_i^{(p)}) g_j^{(f)}(X_i^{(p)}) + \phi(X_i^{(p)})$ where $\theta(X_i^{(p)})$ and $\phi(X_i^{(p)})$ are outputs of the neural network (i.e.: $N(X_i^{(p)})$).

In order to achieve an interpretable stacked solution, we follow Breiman's suggestion and consider a second approach to estimate θ_i's which consists in minimizing $R(G_\theta)$ under the constrain that θ_i's are weights, that is, $\theta_i(\mathbf{x}) \geq 0$ and $\sum_{i=1}^k \theta_i(\mathbf{x}) = 1$. Unfortunately, it is challenging to directly impose this restriction to the solution of the neural network. Instead, we use a different parametrization of the problem, which is motivated by Theorem 2.3.1.

Theorem 2.3.1. The solution of

$$\arg\min_{\theta_x} R(G_\theta)$$

under the constrain that $\theta_i(\mathbf{x}) \geq 0$ and $\sum_{i=1}^k \theta_i(\mathbf{x}) = 1$ is given by

$$\theta_\mathbf{x} = \frac{\mathbb{M}_\mathbf{x}^{-1}\mathbf{e}}{\mathbf{e}'\mathbb{M}_\mathbf{x}^{-1}\mathbf{e}}, \tag{2.3.1}$$

where $\mathbf{e}$ is a k-dimensional vector of ones and

$$\begin{aligned}\mathbb{M}_{\mathbf{x}} &= [\mathbf{E}\left[(Y-g_i(\mathbf{x}))(Y-g_j(\mathbf{x}))\middle|\mathbf{X}=\mathbf{x}\right]_{ij}] \\ &= \mathbf{E}[Y^2|\mathbf{x}] - \mathbf{E}[Y|\mathbf{x}](g_i(\mathbf{x})-g_j(\mathbf{x}))) + g_i(\mathbf{x})g_j(\mathbf{x}).\end{aligned}$$

with $(i,j) \in \{1,...,k\}^2$.

Proof. Notice that

$$R(G_\theta) = \mathbf{E}\left[\mathbf{E}\left[(Y-G_\theta(\mathbf{X}))^2\middle|\mathbf{X}\right]\right].$$

Hence, in order to minimize $R(G_\theta)$, it suffices to minimize $\mathbf{E}\left[(Y-G_\theta(\mathbf{x}))^2\middle|\mathbf{X}=\mathbf{x}\right]$ for each $\mathbf{x} \in \mathscr{X}$. Now, once $\sum_{i=1}^k \theta_i(x) = 1$, it follows that,

$$\begin{aligned}\mathbf{E}\left[(Y-G_\theta(\mathbf{x}))^2|\mathbf{X}=\mathbf{x}\right] &= \mathbf{E}\left[\left(\sum_{i=1}^k \theta_i(\mathbf{x})(Y-g_i(\mathbf{x}))\right)^2 |\mathbf{X}=\mathbf{x}\right] \\ &= \sum_{i,j} \theta_i(\mathbf{x})\theta_j(\mathbf{x})\mathbf{E}\left[(Y-g_i(\mathbf{x}))(Y-g_j(\mathbf{x}))|\mathbf{X}=\mathbf{x}\right] \\ &= \theta_{\mathbf{x}}^t \mathbb{M}_{\mathbf{x}} \theta_{\mathbf{x}},\end{aligned}$$

where $\theta_{\mathbf{x}} = (\theta_1(\mathbf{x}),\ldots,\theta_k(\mathbf{x}))'$. Using Lagrange multipliers, the optimal weights can by found by minimizing

$$f(\theta_{\mathbf{x}},\lambda) := \theta_{\mathbf{x}}^t \mathbb{M}_{\mathbf{x}} \theta_{\mathbf{x}} - \lambda(\mathbf{e}^t\theta_{\mathbf{x}} - 1). \tag{2.3.2}$$

Now,

$$\frac{\partial f(\theta_{\mathbf{x}},\mathbf{x})}{\partial \theta_{\mathbf{x}}} = 2\mathbb{M}_{\mathbf{x}}\theta_{\mathbf{x}} - \lambda\mathbf{e},$$

and therefore the optimal solution satisfies $\theta_{\mathbf{x}}^* = \frac{\lambda}{2}\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e}$. Substituting this on Equation 2.3.2 , obtain that

$$f(\theta_{\mathbf{x}}^*,\lambda) = -\frac{\lambda^2}{4}\mathbf{e}^t\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e} + \lambda,$$

and hence

$$\frac{\partial f(\theta_{\mathbf{x}}^*,\lambda)}{\partial \lambda} = 0 \iff \lambda = \frac{2}{\mathbf{e}^t\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e}},$$

which yields the optimal solution

$$\theta_{\mathbf{x}}^* = \frac{\lambda}{2}\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e} = \frac{2}{\mathbf{e}^t\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e}}\frac{1}{2}\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e} = \frac{\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e}}{\mathbf{e}^t\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e}}.$$

□

Theorem 2.3.1 shows that, under the given constrains, $\theta(\mathbf{x})$ is uniquely defined by $\mathbb{M}_{\mathbf{x}}^{-1}$. Now, because $\mathbb{M}_{\mathbf{x}}$ is a covariance matrix, then $\mathbb{M}_{\mathbf{x}}^{-1}$ is positive definite, and thus Cholesky decomposition can be applied to it. It follows that $\mathbb{M}_{\mathbf{x}}^{-1} = L_{\mathbf{x}}L_{\mathbf{x}}'$, where $L_{\mathbf{x}}$ is a lower triangular matrix. This suggests that we estimate $\theta_{\mathbf{x}}$ by first estimating $L_{\mathbf{x}}$ and then plugging the estimate back into Equation 2.3.1. That is, in order to obtain a good estimator

under the above mentioned restrictions, the output of the network is set to be $L_{\mathbf{x}}$ rather than the weights themselves[1]. We name this method *Constrained Neural Network Stacking* (CNNS). Figure 5 illustrates a CNNS that stacks 2 base regressors (that is, $L_{\mathbf{x}} = [l_{ij}]$ is a $2x2$ triangular matrix) in a 4 feature regression problem.

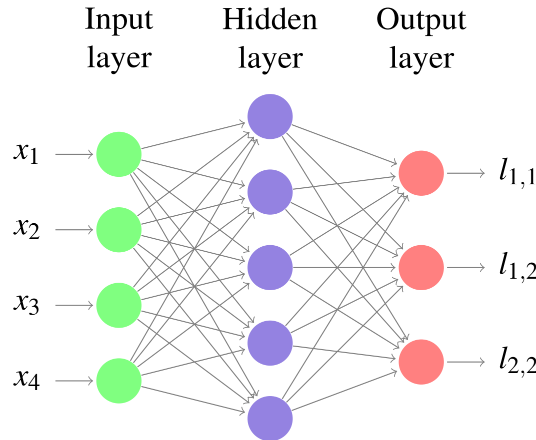

Figure 5 – Example of the CNNS neural network.

Algorithm 3 shows the implementation of this method. As with UNNS, we also explore a variation which adds an extra network output $\phi(\mathbf{x})$ to G_θ.

Algorithm 3 – CNNS

Input: Estimation algorithms $g = (g_1, g_2, \ldots, g_k)'$, a dataset $D = (X, Y)$ with n instances (rows), a neural network N, features to predict $X^{(p)}$, the amount of folds F.

Output: Predicted values $y^{(p)}$.

1: Follow steps 1 to 4 from algorithm 2.

2: Train the neural network N with each input instance i given by a row of X; with a lower triangular matrix L_{X_i} and a scalar $\phi(X_i)$ as outputs; and with loss function given by $(\sum_{j=1}^{k} \theta_j(X_i) P_{ij} + \phi(X_i) - y_i)^2$, where $\mathbb{M}^{-1} = L_{X_i} L'_{X_i}$; $\theta(X_i) = \frac{\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e}}{\mathbf{e}'\mathbb{M}_{\mathbf{x}}^{-1}\mathbf{e}}$ and $\mathbf{e}$ is a k-dimensional vector of ones (note: the additional scalar ϕ is optional, i.e.: it can be set to zero).

3: The predicted value $Y^{(p)}$ is calculated analogously to Algorithm 2.

Figure 6 illustrates the full training process. For simplicity, the neural network early stopping patience criterion is set to a single epoch and the additional parameter $\phi_{\mathbf{x}}$ is not used.

2.3.1 Comparison with standard stacking methods

Most stacking methods create a meta-regression model by applying a regression method directly on the outputs of individual predictions. In particular, a meta-regression method can be a neural network. Such procedure differs from NN-Stacking by the shape

[1] Since the gradients for all matrix operations are implemented for Pytorch tensor classes, the additional operations of the CNNS method will be automatically backpropagated once Pytorch's backward method is called on the loss evaluation.

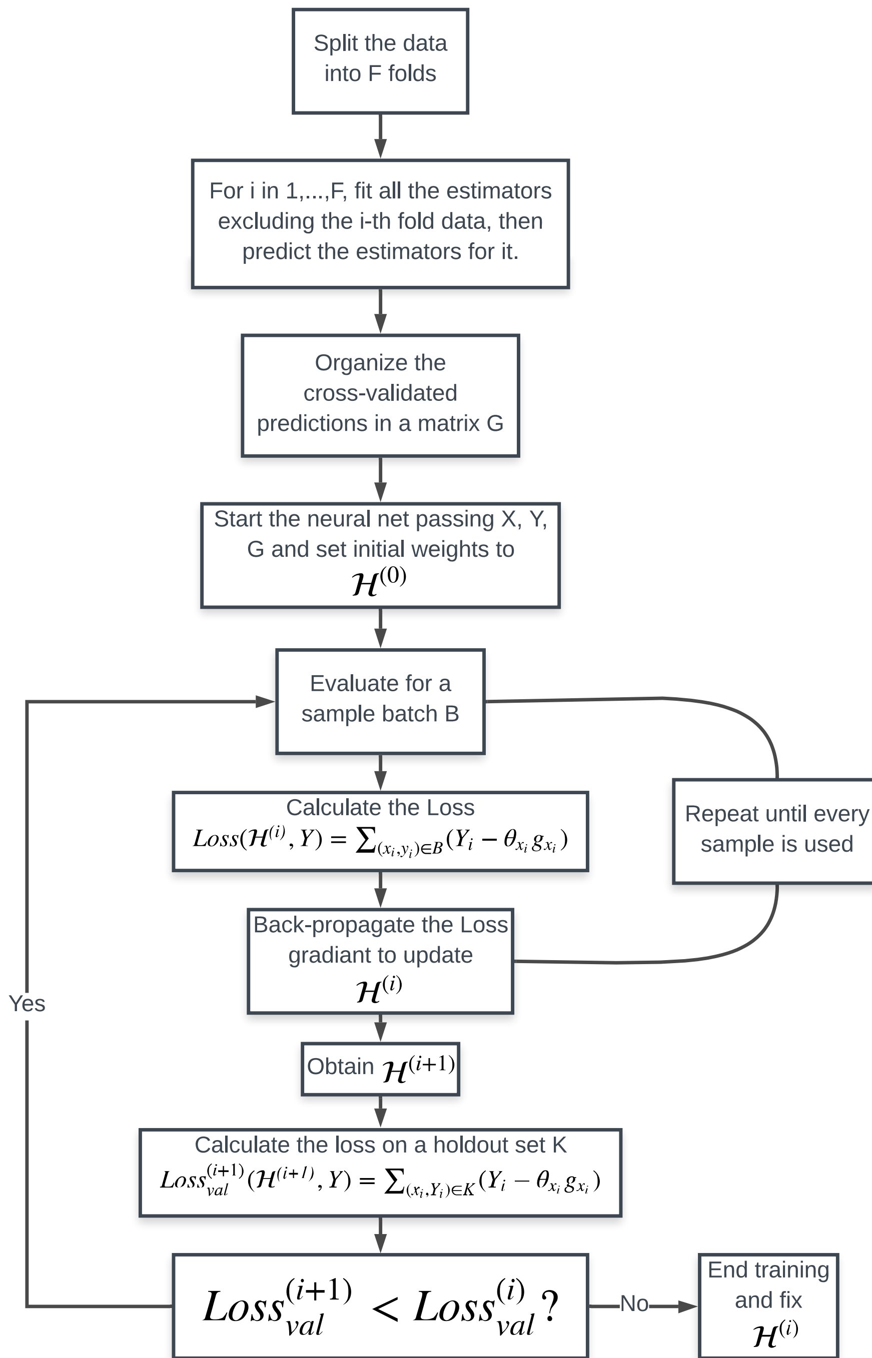

Figure 6 – Full NN-Stacking training process.

of both the input and of the output of the network. While standard stacking uses base regression estimates (g_x) as input and Y as output, NN-Stacking uses the features as input and either the weights θ_x (for UNSS) or $L_{\mathbf{x}}$ (for CNSS) as outputs. The base regression estimates are used only on the loss function. Thus, the NN-Stacking method leads to more interpretable models. Section 2.4 compares these methods in terms of their predictive power. We also point out that our approach has some similarity to Sill *et al.* (2009), which allows each θ_i to depend on meta-features computed from $\mathbf{x}$ using a specific parametric form. Neural networks, on the other hand, provide a richer family of functions to model such dependencies (in fact, they are universal approximators; Csáji (2001)).

2.3.2 Selecting base regressors

Consider the extreme case where $g_i(\mathbf{x}) = g_j(\mathbf{x}) \forall \mathbf{x} \in \mathscr{X}$ for some $i \neq j$, that is, the case in which two base regressors generate the same prediction over all feature space. Now, suppose that one fits a NNS (either CNNS or UNNS) for this case. Then $\theta_i g_i(\mathbf{x}) + \theta_j g_j(\mathbf{x}) = (\theta_i + \theta_j) g_i(\mathbf{x})$. Thus, one of the regressions can be dropped from the stacking with no loss in predictive power.

In practice, our experiments (Section 2.4) show that regression estimators that have strongly correlated results do not contribute to the meta-learner. This suggests that one should choose base regressors with considerably distinct nature.

2.3.3 Implementation details

A Python package that implements the methods proposed in this paper is available at <github.com/randommm/nnstacking>. The scripts for the experiments in Section 2.4 are availiable at <github.com/vcoscrato/NNStacking>. We work with the following specifications for the artificial neural networks:

- **Optimizer**: we use the Adam algorithm (KINGMA; BA, 2014) and decrease its learning rate after the validation loss stops improving for a user-defined number of epochs.
- **Initialization**: we use the Xavier Gaussian method proposed by Glorot and Bengio (2010) to sample the initial parameters of the neural network.
- **Layer activation and regularization**: we use ELU (CLEVERT; UNTERTHINER; HOCHREITER, 2015) as the activation function, and do not use regularization.
- **Normalization**: we use batch normalization (IOFFE; SZEGEDY, 2015) to speed-up the training process.
- **Stopping criterion**: in order to address the risk of having strong over-fit on the neural networks, we worked with a 90%/10% split early stopping for small datasets and a higher split factor for larger datasets (increasing the proportion of training instances) and a patience of 10 epochs without improvement on the validation set.
- **Dropout**: We use dropout (with a rate of 50%) to address the problem of over-fitting (HINTON *et al.*, 2012).
- **Software**: we use PyTorch (PASZKE *et al.*, 2017).
- **Architecture**: as default values we use a 3 layer depth network with hidden layer size set to 100; these values have been experimentally found to be suitable in our experiments (Section 2.4).

2.4 Experiments

We compare stacking methods for the following UCI datasets:

- The GPU kernel performance dataset (241600 instances, 13 features) (NUGTEREN; CODREANU, 2015),
- The music year prediction dataset (DHEERU; TANISKIDOU, 2017) (515345 instances and 90 features),
- The blog feedback dataset (BUZA, 2014) (60021 instances, 280 features),
- The superconductivity dataset (HAMIDIEH, 2018) (21263 instances, 80 features).

First, we fit the following regression estimators (that will be stacked):

- Three linear models: with L1, L2, and no penalization (FRIEDMAN; HASTIE; TIBSHIRANI, 2001),
- Two tree based models: bagging and random forests (FRIEDMAN; HASTIE; TIBSHIRANI, 2001),
- A gradient boosting method (GBR) (MEIR; RÄTSCH, 2003).

The tuning parameters of these estimators are chosen by cross-validation using scikit-learn (PEDREGOSA *et al.*, 2011).

Using these base estimators, we then fit four variations of NNS (both CNNS and UNNS with and without the additional ϕ_x) using the following specifications:

- **Tuning**: Four different architectures were tested for each neural network approach. The layer size was fixed at 100 and the number of hidden layers were set to 1, 3, 5 and 10. We choose the architecture with the lowest validation mean-squared error in the test data.
- **Train/validation/test split**: for all datasets, we use 75% of the instances to fit the models, among which 10% are used for performing early stop. The remaining 25% of the instances are used as a test set to compare the performance of the various models. The train/test split is performed at random. The cross-validated predictions (the matrix P denoted on Algorithm 2) are obtained using a 10-fold cross-validation on the training data (i.e., $F = 10$).
- **Total fitting time**: we compute the total fitting time (in seconds; including the time for cross-validating the network architecture) of each method on two cores of an AMD Ryzen 7 1800X processor running at 3.6Gz with 32GB ram.

We compare our methods with Breiman's linear stacking and the usual neural net stacking model described in Section 2.3.1. In addition to these, we also include a comparison with a direct neural network that has $\mathbf{x}$ as its input and Y as its output.

The comparisons are made by evaluating the mean squared error (MSE, $n^{-1}\sum_{i=1}^{n}(y_i - g(\mathbf{x}_i))^2$) and the mean absolute error (MAE, $n^{-1}\sum_{i=1}^{n}|y_i - g(\mathbf{x}_i)|$) of each model g on a test set. We also compute the standard error for each of these metrics, which enables one to compute confidence intervals for the errors of each method.

2.4.1 GPU kernel performance dataset

Table 1 shows the results that were obtained for the GPU kernel performance dataset. Our UNNS methods outperforms both Breiman's stacking and the usual meta-regression stacking approaches in terms of MSE. Moreover, the UNNS model is also the best one in terms of MAE, even though the gap between the models is lower in this case. Our stacking methods also perform better than all base estimators. This suggests that each base model performs better on a distinct region of the feature space.

Figure 7 shows a boxplot with the distribution of the fitted θ_i's for UNNS. Many fitted values fall out of the range $[0,1]$, which explains why UNNS gives better results than Breiman's and CNNS (which have the restriction that θ_i's must be proper weights).

Table 2 shows the correlation between the prediction errors for base estimators. The linear estimators had an almost perfect pairwise correlation, which indicates that removing up to 2 of them from the stacking would not affect predictions. Indeed, after refitting UNNS without using ridge regression and lasso, we obtain exactly the same results. We also refit the best UNNS removing all of the linear estimators to check if poor performing estimators are making stacking results worse. In this setting, we obtain an MSE of $11074.13(\pm 227.57)$, and a MAE of $45.76(\pm 0.39)$. Note that although the point estimates of the errors are lower than those obtained in Table 1, the confidence intervals have an intersection, which leads to the conclusion that the poor performance of linear estimators is not damaging the stacked estimator.

2.4.2 Music year dataset

Table 3 shows the accuracy metrics results for the music year dataset. In this case, the CNNS gave the best results, both in terms of MSE and MAE. For this dataset, Breiman's stacking was worse than using gradient boosting, one of the base regressors. The same happens with the usual meta-regression neural network approach. On the other hand, NNS could find a representation that combines the already powerful GBR estimator with less powerful ones in a way that leverages their individual performance.

All base estimators had high prediction error correlations (Table 4). In particular, two of the linear estimators could be removed from the stacking without affecting its

Type	Model (Best architecture)	MSE	MAE	Total fit time
Stacked estimators	UNNS + ϕ_x (3 layers)	11400.43 ($\pm$ 250.03)	45.91 ($\pm$ 0.39)	3604
	CNNS + ϕ_x (3 layers)	19371.98 ($\pm$ 429.96)	53.09 ($\pm$ 0.52)	3531
	UNNS (3 layers)	11335.85 ($\pm$ 241.94)	45.85 ($\pm$ 0.39)	3540
	CNNS (3 layers)	18748.66 ($\pm$ 424.5)	51.65 ($\pm$ 0.52)	3387
	Breiman's stacking	30829.11 ($\pm$ 717.13)	62.41 ($\pm$ 0.67)	63
	Meta-regression neural net (10 layers)	24186.4 ($\pm$ 545.52)	58.79 ($\pm$ 0.59)	85
Direct estimator	Direct neural net (10 layers)	14595.98 ($\pm$ 307.11)	52.3 ($\pm$ 0.44)	380
Base estimators	Least squares	79999.09 ($\pm$ 1504.75)	176.41 ($\pm$ 0.9)	-
	Lasso	80091.85 ($\pm$ 1526.05)	175.5 ($\pm$ 0.9)	-
	Ridge	79999.05 ($\pm$ 1504.76)	176.41 ($\pm$ 0.9)	-
	Bagging	31136.93 ($\pm$ 737.47)	62.35 ($\pm$ 0.67)	-
	Random forest	30923.64 ($\pm$ 727.99)	62.2 ($\pm$ 0.67)	-
	Gradient boosting	32043.23 ($\pm$ 676.1)	90.51 ($\pm$ 0.63)	-

Table 1 – Evaluation of model accuracy metrics for the GPU kernel performance dataset.

Models	Least squares	Lasso	Ridge	Bagging	Random forest	Gradient boosting
Least squares	1.00	1.00	1.00	0.39	0.39	0.80
Lasso	1.00	1.00	1.00	0.39	0.39	0.80
Ridge	1.00	1.00	1.00	0.39	0.39	0.80
Bagging	0.39	0.39	0.39	1.00	0.98	0.62
Random forest	0.39	0.39	0.39	0.98	1.00	0.62
Gradient boosting	0.80	0.80	0.80	0.62	0.62	1.00

Table 2 – Pearson correlation between base estimators prediction errors for the GPU kernel performance dataset.

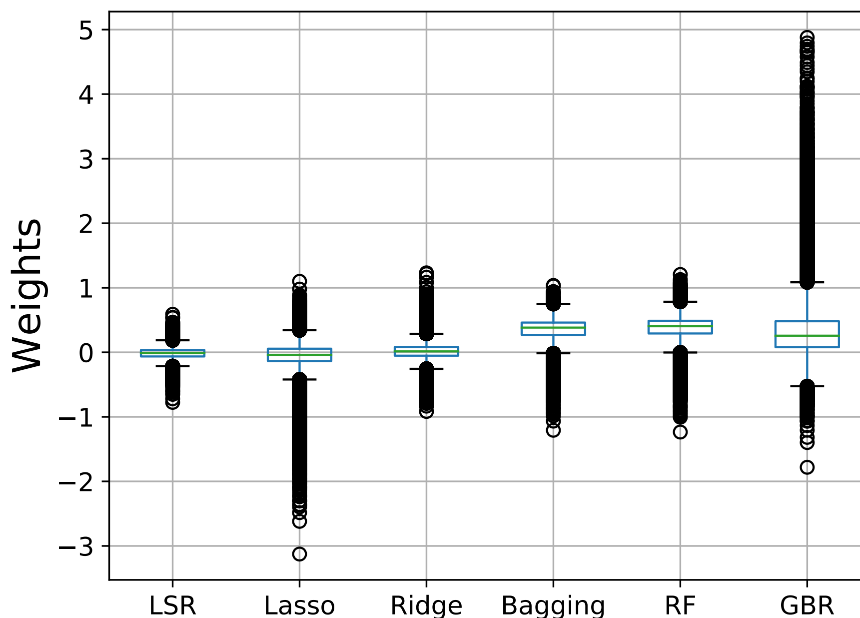

Figure 7 – Weight distribution for the GPU kernel performance dataset.

performance. However, when removing all three linear estimators the MSE for the best NNS increased to 83.92($\pm$0.57) and its MAE increased to 6.44($\pm$0.02).

Figure 8 shows that the fitted NNS weights have a large dispersion. This illustrates the flexibility added by our method. Models with very distinctive nature (e.g., ridge regres-

sion - which imposes a linear shape on the regression function, and random forests - which is fully non-parametric) can add to each other, getting weights of different magnitudes depending on the region of the feature space that the new instance lies on.

Type	Model (Best architecture)	MSE	MAE	Total fit time
Stacked estimators	UNNS + $\phi_{\mathbf{x}}$ (10 layers)	92.37 ($\pm$ 7.18)	6.53 ($\pm$ 0.02)	9432
	CNNS + $\phi_{\mathbf{x}}$ (3 layers)	83.05 ($\pm$ 0.57)	6.38 ($\pm$ 0.02)	8851
	UNNS (10 layers)	95.35 ($\pm$ 1.81)	7.45 ($\pm$ 0.02)	12087
	CNNS (3 layers)	82.99 ($\pm$ 0.57)	6.38 ($\pm$ 0.02)	11466
	Breiman's stacking	87.66 ($\pm$ 0.57)	6.61 ($\pm$ 0.02)	3090
	Meta-regression neural net (1 layer)	87.64 ($\pm$ 0.59)	6.61 ($\pm$ 0.02)	571
Direct estimator	Direct neural net (1 layer)	1596.2 ($\pm$ 10.88)	29.83 ($\pm$ 0.07)	2341
Base estimators	Least squares	92.03 ($\pm$ 0.62)	6.82 ($\pm$ 0.02)	-
	Lasso	92.61 ($\pm$ 0.62)	6.87 ($\pm$ 0.02)	-
	Ridge	92.03 ($\pm$ 0.62)	6.82 ($\pm$ 0.02)	-
	Bagging	92.83 ($\pm$ 0.59)	6.84 ($\pm$ 0.02)	-
	Random forest	92.6 ($\pm$ 0.59)	6.83 ($\pm$ 0.02)	-
	Gradient boosting	87.49 ($\pm$ 0.6)	6.58 ($\pm$ 0.02)	-

Table 3 – Evaluation of the model accuracy metrics for the music year dataset.

Models	Least squares	Lasso	Ridge	Bagging	Random forest	Gradient boosting
Least squares	1.00	1.00	1.00	0.87	0.87	0.95
Lasso	1.00	1.00	1.00	0.88	0.88	0.96
Ridge	1.00	1.00	1.00	0.87	0.87	0.95
Bagging	0.87	0.88	0.87	1.00	0.89	0.91
Random forest	0.87	0.88	0.87	0.89	1.00	0.91
Gradient boosting	0.95	0.96	0.95	0.91	0.91	1.00

Table 4 – Pearson correlation between base estimators prediction errors for the music year dataset.

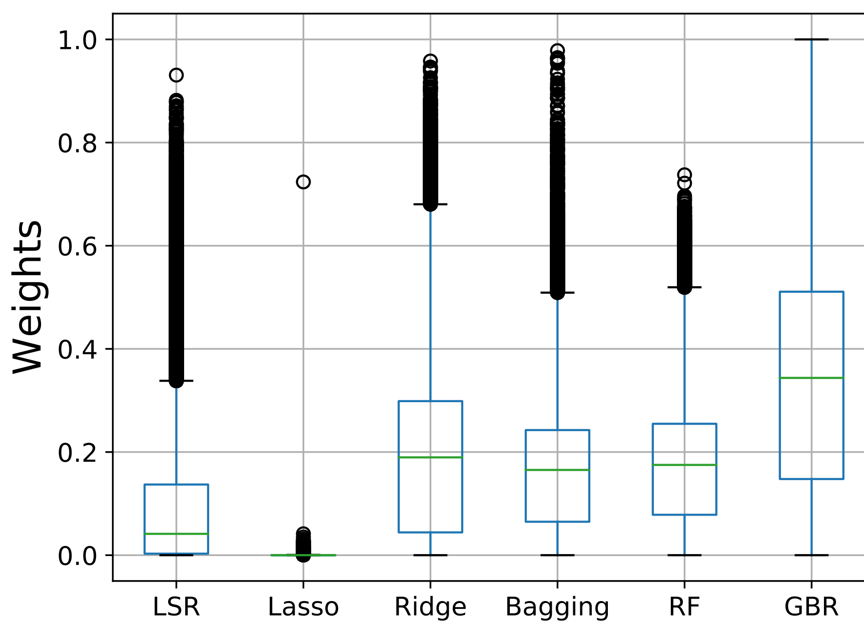

Figure 8 – Weight distribution for the music year dataset.

2.4.3 Blog feedback dataset

Table 5 shows the results for the blog feedback dataset. All stacked estimators had similar performance in terms of MSE. However, UNNS had slightly worse performance with respect to MAE. This may happen because the NNS is designed to minimize the MSE and not the MAE. Overall, for this small dataset, the NNS shows no improvement over Breiman's stacking or the usual meta-regression neural network.

GBR had the lowest MSE for the base estimators, while bagging and random forests had the lowest MAE. This explains why these models have larger fitted weights (Figure 9). Moreover, the linear models prediction errors had an almost perfect error correlation (Table 6). This suggests that removing up to 2 of them from the NNS would not impact its performance. Also, the linear estimators has a poor performance when compared to the other base regressors. We thus refit the best NNS for this data after removing these estimators, and achieve an MSE of 531.88 (±62.67) and a MAE of 5.31 (±0.20). We conclude that the linear estimators did not damage nor improved the NNS.

Type	Model (Best architecture)	MSE	MAE	Total fit time
Stacked estimators	UNNS + $\phi_\mathbf{x}$ (10 layers)	542.02 (± 62.65)	5.89 (± 0.2)	420
	CNNS + $\phi_\mathbf{x}$ (1 layer)	548.99 (± 63.9)	5.44 (± 0.2)	404
	UNNS (10 layers)	557.95 (± 61.51)	6.38 (± 0.2)	447
	CNNS (3 layers)	540.68 (± 63.87)	5.44 (± 0.2)	433
	Breiman's stacking	593.74 (± 73.19)	5.41 (± 0.21)	202
	Meta-regression neural net (3 layers)	537.66 (± 63.31)	5.53 (± 0.2)	44
Direct estimator	Direct neural net (3 layers)	676.79 (± 81.0)	7.52 (± 0.22)	63
Base estimators	Least squares	878.88 (± 109.42)	9.56 (± 0.25)	-
	Lasso	877.11 (± 108.11)	9.04 (± 0.25)	-
	Ridge	877.92 (± 109.47)	9.53 (± 0.25)	-
	Bagging	619.04 (± 88.49)	5.27 (± 0.21)	-
	Random forest	585.22 (± 64.88)	5.37 (± 0.21)	-
	Gradient boosting	557.28 (± 63.88)	5.75 (± 0.2)	-

Table 5 – Evaluation of model accuracy metrics for the blog feedback dataset.

Models	Least squares	Lasso	Ridge	Bagging	Random forest	Gradient boosting
Least squares	1.00	0.99	1.00	0.68	0.70	0.81
Lasso	0.99	1.00	0.99	0.68	0.69	0.81
Ridge	1.00	0.99	1.00	0.68	0.70	0.81
Bagging	0.68	0.68	0.68	1.00	0.92	0.89
Random forest	0.70	0.69	0.70	0.92	1.00	0.90
Gradient boosting	0.81	0.81	0.81	0.89	0.90	1.00

Table 6 – Pearson correlation between base estimators prediction errors for the blog feedback dataset.

2.4.4 Superconductivity dataset

The results for the superconductivity dataset (Table 7) were similar to those obtained for the blog feedback data: the NNS methods perform slightly better than Breiman's

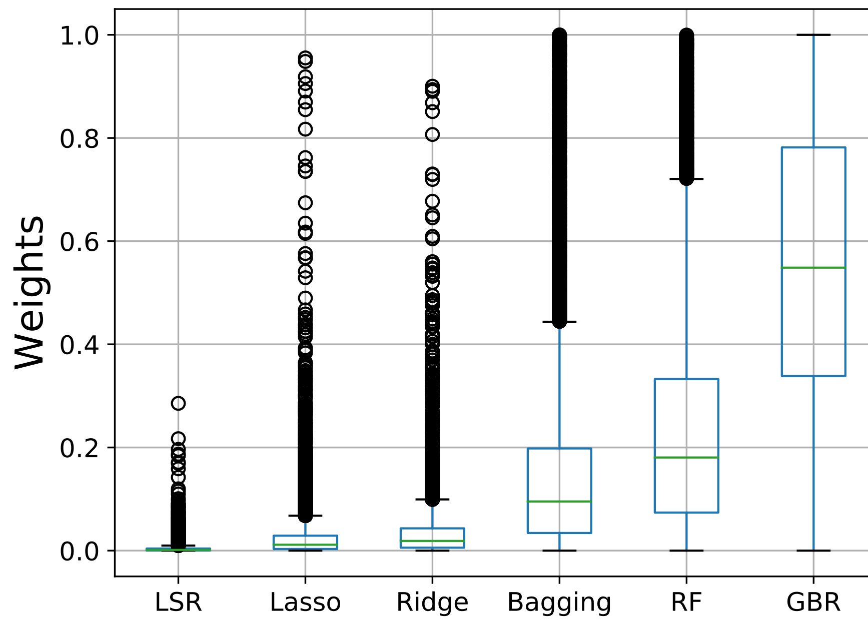

Figure 9 – Weight distribution for the blog feedback dataset.

in terms of MSE, and worse in terms of MAE. Moreover, both tree-based models had the best MSE among base estimators, competing with the GBR in terms of MAE. Hence, they got larger fitted weights (Figure 10).

Table 8 shows that GBR did not have a high correlation error to the tree-based estimators (0.72 in both cases). This is another reason why although having higher MSE, the GBR has high fitted weight for some instances. One can also note that bagging and random forest had an almost perfect error correlation. This implies that removing one of them would lead to no changes in the NNS. Finally, removing the linear models did not change the MSE and the MAE for the stacking methods.

Type	Model (Best architecture)	MSE	MAE	Total fit time
Stacked estimators	UNNS + $\phi_{\mathbf{x}}$ (10 layers)	98.97 ($\pm$ 4.67)	5.71 ($\pm$ 0.11)	334
	CNNS + $\phi_{\mathbf{x}}$ (1 layer)	98.79 ($\pm$ 4.67)	5.65 ($\pm$ 0.11)	325
	UNNS (10 layers)	98.62 ($\pm$ 4.77)	5.64 ($\pm$ 0.11)	344
	CNNS (3 layers)	98.60 ($\pm$ 4.75)	5.60 ($\pm$ 0.11)	335
	Breiman's stacking	99.79 ($\pm$ 4.95)	5.48 ($\pm$ 0.11)	48
	Meta-regression neural net (1 layer)	99.05 ($\pm$ 4.78)	5.60 ($\pm$ 0.11)	24
Direct estimator	Direct neural net (3 layers)	274.93 ($\pm$ 7.20)	7.20 ($\pm$ 0.16)	62
Base estimators	Least squares	308.65 ($\pm$ 13.41)	7.12 ($\pm$ 0.16)	-
	Lasso	475.6 ($\pm$ 17.08)	9.41 ($\pm$ 0.19)	-
	Ridge	309.17 ($\pm$ 13.42)	7.17 ($\pm$ 0.16)	-
	Bagging	105.14 ($\pm$ 5.68)	5.02 ($\pm$ 0.12)	-
	Random forest	103.02 ($\pm$ 5.59)	5.08 ($\pm$ 0.12)	-
	Gradient boosting	161.48 ($\pm$ 8.74)	5.05 ($\pm$ 0.13)	-

Table 7 – Evaluation of model accuracy metrics for the superconductivity dataset.

Models	Least squares	Lasso	Ridge	Bagging	Random forest	Gradient boosting
Least squares	1.00	0.80	1.00	0.52	0.51	0.78
Lasso	0.80	1.00	0.80	0.45	0.44	0.67
Ridge	1.00	0.80	1.00	0.52	0.51	0.78
Bagging	0.52	0.45	0.52	1.00	0.91	0.72
Random forest	0.51	0.44	0.51	0.91	1.00	0.72
Gradient boosting	0.78	0.67	0.78	0.72	0.72	1.00

Table 8 – Pearson correlation between base estimators prediction errors for the superconductivity dataset.

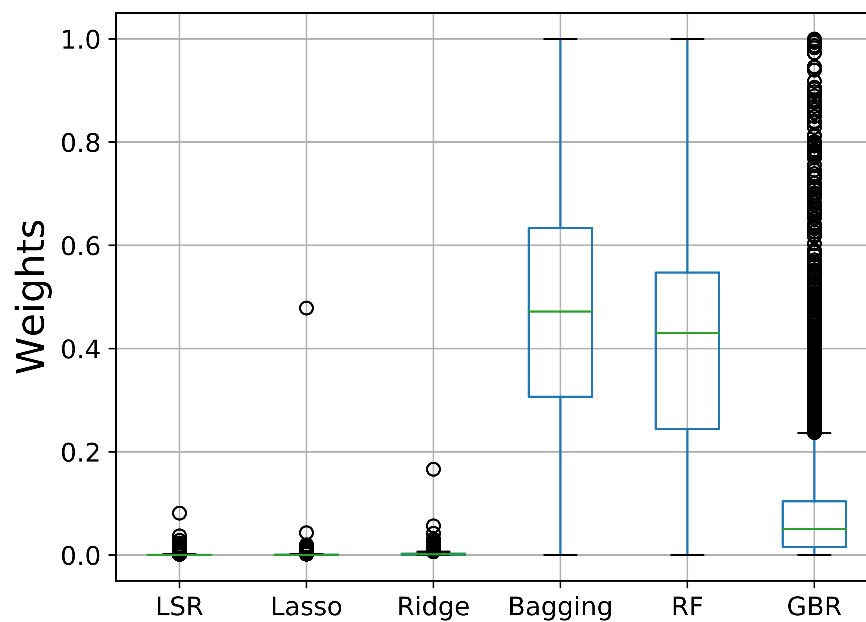

Figure 10 – Weight distribution for the superconductivity dataset.

CHAPTER 3

NLS

Machine learning applications are often focused on maximizing prediction accuracy, leading practitioners to choose highly complex regression estimators (VACH; ROßNER; SCHUMACHER, 1996). In this scenario, neural networks have recently gained much prominence in regression applications due to its high predictive accuracy and its scalability to large datasets.

However, in many applications, accuracy is only one of the features that must be considered when choosing which prediction method to use. Another relevant aspect is the easiness in interpreting the outputs of a method at hand. The ability to explain predictions made by a method is important to give insights about the decision making learned by the model; that can increase the trust practitioners have over the ML model (DOSHI-VELEZ; KIM, 2017). For this aim, many different approaches have been proposed (e.g. Hechtlinger (2016) and references therein), these model interpretation techniques are far from consensual.

One solution to such issue is using model agnostic interpreters, such as LIME (RIBEIRO; SINGH; GUESTRIN, 2016). LIME uses a kernel smoother to fit a local linear approximation to a (possibly complex) regression function around the instance to be explained. By looking at the coefficients of this approximation, it is then possible to explain why such prediction was made.

LIME and related methods work on demand. That is, every time a new instance, $\mathbf{x}^*$, needs to be explained, a local linear estimator is fit on a neighborhood around $\mathbf{x}^*$. This can be too slow to be applied in practice. Moreover, several nontrivial choices on how to define the neighborhood around $\mathbf{x}^*$ and how sample from it need to be made (BOTARI; IZBICKI; CARVALHO, 2019). Thus, this two-step approach of first learning the network and then explaining it is not practical.

In this work we introduce the Neuro Local Smoother (NLS), a one-step approach to fit a neural network that yields predictions that are easy to be explained. The key idea of

the method is to combine the architecture of the network with a local linear output. We show that NLS maintains the high predictive accuracy of the neural networks while being, by default, highly interpretable, with no need for using external model interpreters.

3.1 The NLS

Consider a set of data instances $(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)$, where $\mathbf{X}_i \in \mathbb{R}^d$ are features and $Y_i \in \mathbb{R}$ is the label to be predicted. The Neural Local Smoother learns a neural network that ensures a local linear shape to the prediction function. In order to do so, this neural network has input $\mathbf{x}_{(dx1)}$ and output $\Theta(\mathbf{x}) = (\theta_0, \theta_1(\mathbf{x}), \ldots, \theta_d(\mathbf{x}))$. An example of a NLS network containing 4 features and a single hidden layer with 4 neurons is shown in Figure 11. In order to obtain the predictions, these outputs are then combined according to

$$G_\Theta(\mathbf{x}) := \theta_0 + \sum_{i=1}^{d} \theta_i(\mathbf{x}) x_i. \tag{3.1.1}$$

The prediction function of Equation 3.1.1 is easy to interpret because it is locally linear. Thus, given a new instance $\mathbf{x}^*$, one can interpret the prediction made to $\mathbf{x}^*$ by looking at the coefficients $\theta_i(\mathbf{x}^*)$ in a similar way as done in LIME.

Consider a fixed architecture of a neural network that maps $\mathbf{x} \in \mathbf{R}^d$ into $\Theta(\mathbf{x}) \in \mathbf{R}^{d+1}$. Let Γ be the set of all possible values for the parameters (weights) associated to that network. Each $\gamma \in \Gamma$ is then associate to a different choice of $\Theta(\mathbf{x})$. In order to learn the weights of the network, the NLS uses a squares loss function over a given training dataset $(x_1, y_1), \ldots, (x_n, y_n)$, that is,

$$\gamma^* = \arg\min_{\gamma \in \Gamma} \sum_{i=1}^{n} \left[(y_i - G_\Theta(\mathbf{x}_i))^2 \right] \tag{3.1.2}$$

As long as the architecture of the network is sufficiently complex, any regression function can be represented by Equation 3.1.1. This is shown in the following theorem:

Theorem 3.1.1. Let $r(\mathbf{x}) := \mathbf{E}[Y|\mathbf{x}]$ be the true regression function and let $\varepsilon > 0$. Assume that the domain of the feature space is $[0,1]^d$. If $r(\mathbf{x})$ is continuous, then there exists an architecture and weights for NLS such that $|r(\mathbf{x}) - G_\Theta(\mathbf{x})| < \varepsilon$ for every $\mathbf{x} \in (0,1)^d$.

Proof. Because $h(\mathbf{x}) := \frac{r(\mathbf{x})}{x_1}$ is also continuous, it follows from the universal representation theorem (CYBENKO, 1989) that there exists a neural network with output $N(\mathbf{x})$ such that

$$|h(\mathbf{x}) - N(\mathbf{x})| < \varepsilon$$

for every $\mathbf{x} \in (0,1)^d$.

Now, let $\theta_1(\mathbf{x}) = N(\mathbf{x})$, $\theta_0 = 0$, and $\theta_i(\mathbf{x}) \equiv 0$ for every $i > 1$, and thus $G_\Theta(\mathbf{x}) = \theta_1(\mathbf{x})x_1$. Because $0 < x_1 < 1$, it follows that

$$|r(\mathbf{x}) - G_\Theta(\mathbf{x})| = |r(\mathbf{x}) - N(\mathbf{x})x_1| \leq |\frac{r(\mathbf{x})}{x_1} - N(\mathbf{x})| = |h(\mathbf{x}) - N(\mathbf{x})| < \varepsilon,$$

which concludes the proof. □

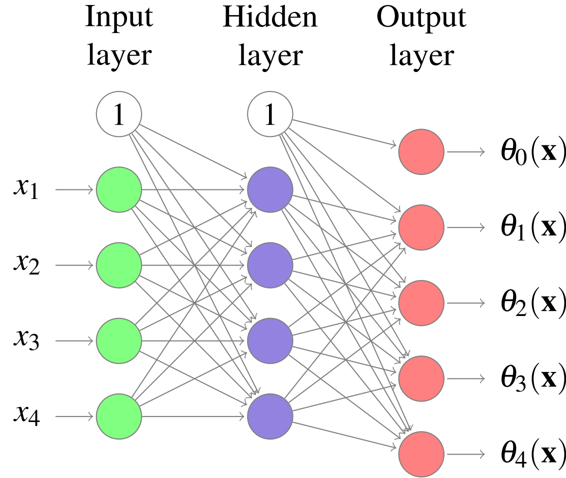

Figure 11 – Example of a NLS neural network.

Theorem 3.1.1 implies that, for a complex enough architecture, an NLS can fully represent any neural network regression. Furthermore, fox a fixed index $i \in 1,\ldots,d$, a NLS with $\theta_j(\mathbf{x}) \equiv 0 \forall j \neq i$ can still fully represent any neural network regression. Hence, there are infinite choices of $\Theta(\mathbf{x})$, and thus infinite possible NLS adjusts, that leads to the same predictions. In other words, the solution of Equation 3.1.2 is not unique. There might be, therefore, a variety of γ settings leading to similar predictive errors.

As NLS is a local linear method, $\theta_i(\mathbf{x}^*)$ can be locally interpreted as a linear coefficient. Ribeiro, Singh and Guestrin (2018) argue that practitioners tend to extend local interpretation to new samples, which can lead to poor inference as $\theta_i(\mathbf{x})$ varies. Thus, ideally, $\theta_i(\mathbf{x})$ should vary smoothly with $\mathbf{x}$. Therefore, we define an alternative loss function that penalizes non-smooth solutions. We choose γ as follows:

$$\gamma^* = \arg\min_{\gamma \in \Gamma} \sum_{i=1}^{n} \left[(y_i - G_\gamma(\mathbf{x}_i))^2 + \lambda \sum_{k,l \geq 0} \left(\frac{\partial \theta_k(\mathbf{x})}{\partial x(l)} \Big|_{\mathbf{x}=\mathbf{x}_i} \right)^2 \right] \tag{3.1.3}$$

where λ is the penalization strength. These penalization guarantees that the optimization algorithm pursuits γ settings in which Θ variation is smoother, leading to more accurate inferences to new samples when interpretations are extended.

Equation 3.1.3 defines a global interpretability-accuracy trade-off. If $\lambda = 0$, $\Theta(\mathbf{x})$ can vary freely, which is typically lead to predictive models with better accuracy. As $\lambda \longrightarrow \infty$, we recover a plain least squares linear regression (i.e., constant θ_i's), which is highly interpretable, but has low prediction power in most cases. Notice that high values of

λ encourage simpler NLS, which tend to increase model bias while decreasing its variance. Thus, accuracy may also increase with λ.

The approach of introducing a penalty that encourages explainability in prediction methods has also been proposed by Plumb *et al.* (2019) in a general framework. In our case, we choose a regularizer that is particularly suitable to a neural network because it is easy to compute. This is because the derivatives in Equation 3.1.3 come straight up from the back-propagation algorithm on the network fit.

Example 3.1.2 presents a toy experiment to show the interpretability-accuracy trade-off in practice.

Example 3.1.2. In this example, we use a NLS to fit the function $y = \sin(x)$ in the interval $[0, 2\pi]$. For this, we sampled 2.000 points in this interval and adjusted the NLS for λ varying in $[0, 2]$ using 80% of the points (randomly selected). For the remaining 20%, we obtained the MSE and the average squared gradient (in this case, as x in uni-dimensional, this is the cumulative squared derivative) as function of λ. Figure 12 illustrates the obtained results.

Remark 1. *This penalized approach has some similarity to ridge regression (HOERL; KENNARD, 1970) and lasso (TIBSHIRANI, 1996). On ridge regresison, penalization encourages search for shrinked solutions, while on lasso, penalization leads to shrinked and sparse solutions. Here, the penalty addition leads to* $\Theta(\mathbf{x})$ *estimates that are are better extensible for new samples.*

Remark 2. *This penalty design leads to a globally smoother choice of* θ*. There is no guarantee, however, of higher smoothness on any specific regions of the feature space.*

3.1.1 Implementation details

The Python package that implements the methods proposed in this paper is available at <github.com/randommm/nnlocallinear>. We work with the following specifications for the artificial neural networks:

- **Optimizer**: we choose to work with the Adam optimizer (KINGMA; BA, 2014) and decrease its learning rate once no improvement can is seem on the validation loss for a considerable number of epochs.
- **Initialization**: to initialize the network weights, we used a method of initialization proposed by Glorot and Bengio (2010).
- **Layer activation and regularization**: we chose ELU (CLEVERT; UNTERTHINER; HOCHREITER, 2015) as the activation function and no regularization.

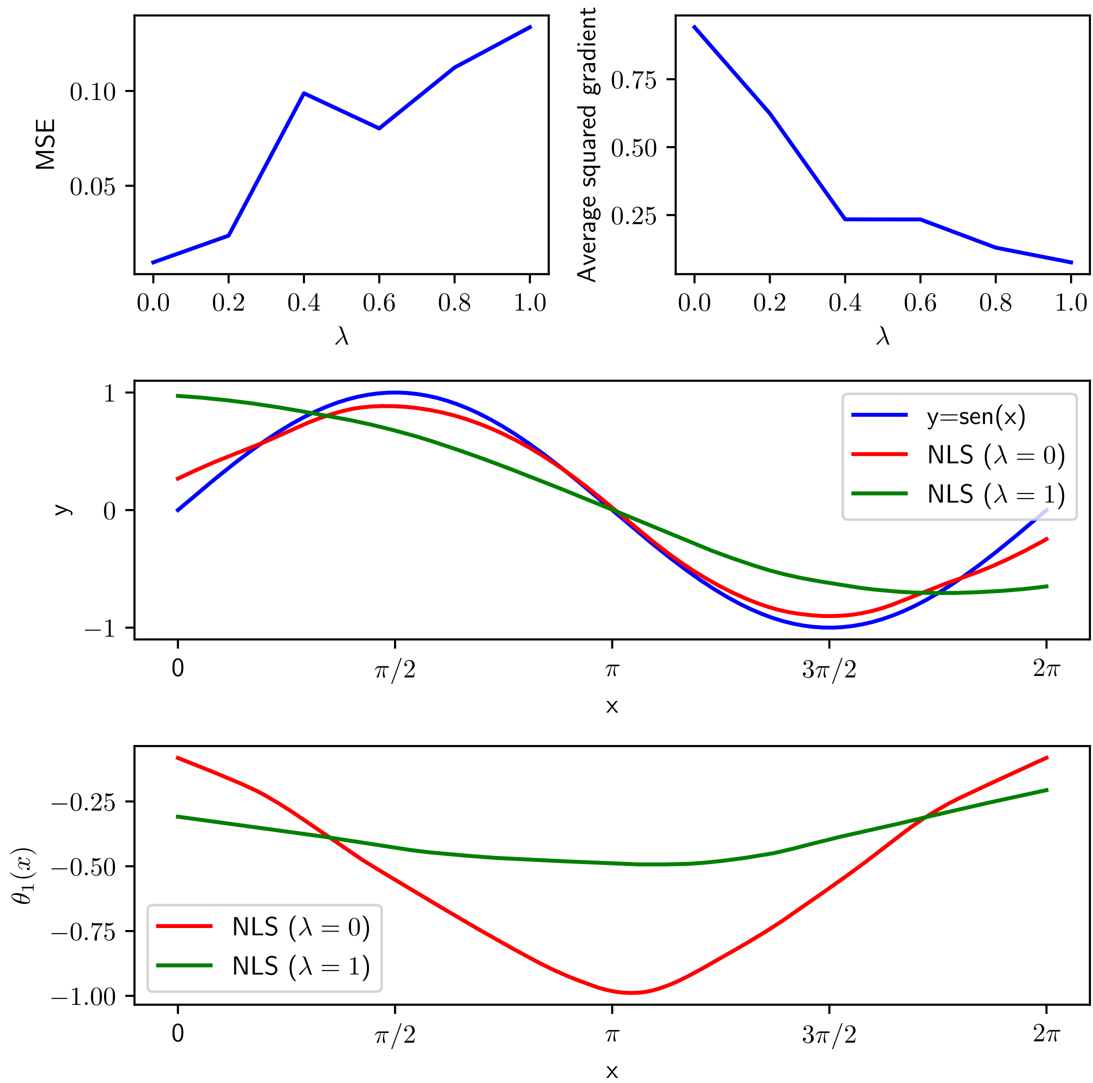

Figure 12 – Top-left: MSE as function of the penalization strength λ. Notice that adding penalization decreases the NLS accuracy, as proposed by the trade-off. Top-right: Average squared gradient as function of λ. It can be seen that higher penalization values lead to smoother estimates for θ. Center and Bottom: True regression, NLS adjusts and fitted $\theta_1(x)$ for $\lambda = 0, 1$. While the non-penalized NLS yields better fit, the penalized version provides a smoother adjust.

- **Stop criterion**: a 90%/10% split early stopping for small datasets and a higher split factor for larger datasets (increasing the proportion of training instances) and a patience of 50 epochs without improvement on the validation set.

- **Normalization**: batch normalization, as proposed by Ioffe and Szegedy (2015), is used in this work in order to speed-up the training process.

- **Dropout**: in this work we also took advantage of dropout which is a technique proposed by Hinton *et al.* (2012).

- **Software**: we have PyTorch as the deep learning framework of choice.

3.1.2 Connection to local linear estimators

Local linear smoothing (LLS) methods (FAN; GIJBELS, 1992; FAN, 1992; FAN, 2018) have shown to be a powerful tool for performing non-parametric regression in several applications (RUAN; FESSLER; BALTER, 2007; MCMILLEN, 2004). Their prediction function has the shape

$$G_\Theta(\mathbf{x}) := \theta_0(\mathbf{x}) + \sum_{i=1}^{d} \theta_i(\mathbf{x})x_i, \tag{3.1.4}$$

that is, LLS also consist in a local linear expression for the regression function. However, rather then estimating the parameters θ_i using neural networks, for each new instance $\mathbf{x}^*$, $\Theta(\mathbf{x}^*) = (\theta_0(\mathbf{x}^*), \ldots, \theta_d(\mathbf{x}^*))$ is estimated using weighted least squares:

$$\hat{\Theta}(\mathbf{x}) = \arg\min_{\theta \in \mathbb{R}^{d+1}} \sum_{i=1}^{n} K(\mathbf{x}, \mathbf{x}_i)(Y_i - \theta_0 - \sum_{i=1}^{d} \theta_i x_i)^2, \tag{3.1.5}$$

where K is a smoothing kernel function. The solution to such minimization problem is given by

$$\hat{\Theta}(\mathbf{x}^*) = (X^T W X)^{-1} X^T W y, \tag{3.1.6}$$

where $W = \text{diag}(K(\mathbf{x}^*, \mathbf{x}_1), K(\mathbf{x}^*, \mathbf{x}_2), ..., K(\mathbf{x}^*, \mathbf{x}_n))$.

Local linear smoothers hold good interpretability properties by default. On the other hand, the LLS calculates pairwise kernels for each new sample, leading to higher calculation effort and memory requirement as training data grows. Also, a new least squares optimization is required for each new sample. Therefore, local smoothers might be slow to generate predictions on high dimensional applications. This is not the case for NLS: once the network is learned, evaluating the prediction on new instances only requires a single feed-forward run though the network.

Moreover, local smoothers have an interpretability concern on θ_0. On linear models, this parameters stands for $E[Y|x = 0]$, but when $\theta_0(\mathbf{x})$ is a function of $\mathbf{x}$, there is not a practical meaning for such parameter. This is why we fix θ_0 on the NLS. The same idea is not directly applicable to local linear estimators due to the multiple optimizations. In this case, some preliminary method to select the fixed value would be needed.

As the kernel controls each training instance weight to generate predictions to a new one, choosing a suitable kernel is important. Ali and Smith-Miles (2006), Khemchandani, Chandra *et al.* (2009), Argyriou *et al.* (2006), Hastie, Loader *et al.* (1993) discuss about kernel usage and algorithms to choose a suitable kernel. These methods rely on some algorithm usage to obtain such kernel. In practice, a family of kernel functions is defined, such as a Gaussian kernel, that is,

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left\{-\frac{d^2(\mathbf{x}_i, \mathbf{x}_j)}{\sigma^2}\right\}$$

where $d(\cdot,\cdot)$ is the euclidean distance and σ a variance parameter that defines how much euclidean proximity impact the kernel weights. With such family, a cross-validation procedure is conduced to choose σ. Unfortunately, choosing a suitable kernel can be slow, once obtaining cross validation predictions is an exhausting process.

Remark 3. *When using a Gaussian kernel on a local linear smoother, the σ kernel hyper-parameter has some relation to our proposed λ on NLS: when $\sigma \longrightarrow \infty$, the weighting is constant among all sample space, and hence, a plain least squares linear regression is recovered. As σ gets smaller, the local linear parameters can vary more.*

Notice that the Gaussian kernel (and so as the most used kernels) does not perform a weighting over data features, that is, every feature is equally relevant to the kernel value. In practice, features do not have the same predictive relevance, hence, an optimal sample weighting procedure should consider feature predictive relevance. The NLS approach defines a local linear estimator that is not dependent on a kernel. Moreover, the universal approximation theorem (HORNIK, 1991) guarantees that any continuous function can be approximated by a complex enough feed forward neural network. In particular, a LLS is a set of continuous functions $\theta_0(\mathbf{x}), \theta_1(\mathbf{x}), \ldots, \theta_d(\mathbf{x})$. Hence, the NLS can represent a local smoother determined by any kernel function. Also, as the neural net has $\mathbf{x}$ as input, the network architecture automatically allows feature selection. Example 3.1.3 shows a simulated example to illustrate this.

Example 3.1.3. Consider the regression model $E(Y|x) = g(x) = x^2$. From this model we sample 2000 instances with $x \in [-5, 5]$. Within this data, we fit NLS (with an architecture of 3 layers of size 500) and a LLS (with a Gaussian kernel, using cross-validation to choose σ). We also add irrelevant features (that is, features that are independent of the label) to the data and refit the models. Figure 13 illustrates the features relationship with Y. Table 9 shows each model mean squared error on a 20% holdout sample. One can notice that NLS suffers small impact from the irrelevant feature addition, contrary to LLS.

Model \ Irrelevant features	0	5	50
NLS	8.64	8.83	13.89
LLS	8.61	473.27	632.83

Table 9 – Models mean scored errors across different irrelevant features number addition. While LLS is heavily affected by irrelevant features due to the issues they cause on sample weighting, NLS does suffer as much.

3.2 Experiments

The NLS is a prediction method that fits a local smoother through neural networks. Therefore, we want to ensure that it gives good predictive accuracy when compared to

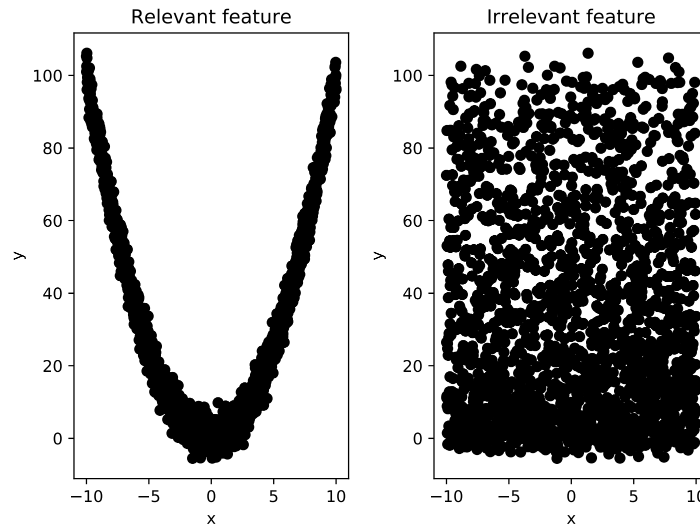

Figure 13 – Relevant and irrelevant features relationship with Y. Notice that the difference in the irrelevant features on different instances does not provide any information about Y. However, a typical kernel is affected in the same way by all features, relevant or irrelevant.

both standard neural network regression and LLS. In this section, we perform comparisons among these models along with random forests (BREIMAN, 2001). We use the following datasets:

- The Boston housing dataset (JR; RUBINFELD, 1978) (506 instances, 13 features),
- The superconductivity dataset (HAMIDIEH, 2018) (21.263 instances, 80 features),
- The blog feedback dataset (BUZA, 2014) (60.021 instances, 280 features),
- The Amazon fine foods dataset (MCAULEY; LESKOVEC, 2013) (100.000 instances, textual data).

For each dataset, we used 80% of the instances to train the models, and the remaining 20% to validate and calculate final model accuracy metrics (for the neural network methods, the early stopping validation set is a 10% part of the training set). We defined validation grid search through the MSE to each technique as follows:

- For the NLS and the neural network regression (NN), we tested using 1, 3, and 5 layers, with sizes 100, 300 and 500 (9 combinations). We used no penalization for the NLS ($\lambda = 0$),
- For the LLS we used a Gaussian kernel and variate the kernel variation parameter in $\{0.1, 1, 10, 100, 1000\}$,
- For the random forests (RF), we used the Scikit-learn (PEDREGOSA *et al.*, 2011) implementation and varied the number of estimators in $\{10, 50, 100\}$.

For the final models obtained, we calculated the MSE, the MAE and both metrics standard deviations. Also, we evaluated the fitting time (in seconds) from every technique (including the cross-validation). These experiments were run on a AMD Ryzen 7 1800X CPU running at 3.6Gz. Table 10 shows the obtained results.

Data	Model	MSE	MAE	Fitting time
Boston housing	NLS	6.03 ($\pm$ 1.00)	1.83 ($\pm$ 0.16)	514
	LLS	8.14 ($\pm$ 1.48)	2.05 ($\pm$ 0.20)	0.20
	NN	8.01 ($\pm$ 1.86)	1.98 ($\pm$ 0.20)	386
	RF	8.44 ($\pm$ 1.76)	1.99 ($\pm$ 0.21)	0.31
Superconductivity	NLS	98.25 ($\pm$ 5.76)	6.00 ($\pm$ 0.12)	10794
	LLS	173.54 ($\pm$ 7.40)	8.36 ($\pm$ 0.16)	1082
	NN	279.03 ($\pm$ 178.09)	12.81 ($\pm$ 0.28)	2198
	RF	84.45 ($\pm$ 5.15)	5.11 ($\pm$ 0.12)	109
Blog feedback	NLS	271.23 ($\pm$ 40.72)	4.98 ($\pm$ 0.16)	14025
	LLS	840.80 ($\pm$ 118.83)	7.10 ($\pm$ 0.27)	38784
	NN	273.81 ($\pm$ 47.67)	4.87 ($\pm$ 0.17)	3622
	RF	256.35 ($\pm$ 36.42)	3.34 ($\pm$ 0.15)	215
Amazon fine foods	NLS	1.07 ($\pm$ 0.02)	0.69 ($\pm$ 0.01)	38754
	LLS	1.14 ($\pm$ 0.02)	0.78 ($\pm$ 0.01)	121371
	NN	1.06 ($\pm$ 0.01)	0.72 ($\pm$ 0.01)	6185
	RF	1.10 ($\pm$ 0.02)	0.70 ($\pm$ 0.01)	601

Table 10 – MSE, MAE and their standard errors for the test set and the fitting times for each dataset. Notice that NLS and the neural network regression have similar predictive accuracy in 2 out of 4 datasets. NLS was more accurate in the Boston housing and the superconductivity data. Also, for every dataset, the results obtained by NLS are similar to the random forests and superior to LLS. The fitting time of NLS is high (especially on high dimensional data), being only overtaken by LLS in the bigger datasets.

It can be noticed that NLS either outperforms or draws against the LLS and NN in all of the datasets. When compared to random forests, NLS was the best in 2 out of 4 datasets. As the NLS is estimated through neural networks, one can expect that its performance is poor on small training sets, the Boston housing data, although, did not confirm such expectation and showed that NLS can perform good fit for small data. NLS and NN fit are similar due to the neural network usage, although, the NLS imposes a more complex output structure to the network, which led to bigger fitting time.

3.2.1 Sample size effect

The Amazon fine foods dataset is a large dataset (> 500.000 instances). In our example, we randomly sampled 100000 instances due to memory lack to fit LLS. We now use the Amazon fine foods dataset to check how the sample size affects both the quality and the fitting time of the models used in our experiments. For this, we selected different size samples from 1.000 to 100.000 from the data. For each data sample, we performed the same experiment described earlier in this section. Figure 14 shows the best

test mean squared error for each method and each sample size and total fit (including cross-validation) time and prediction time for each sample size.

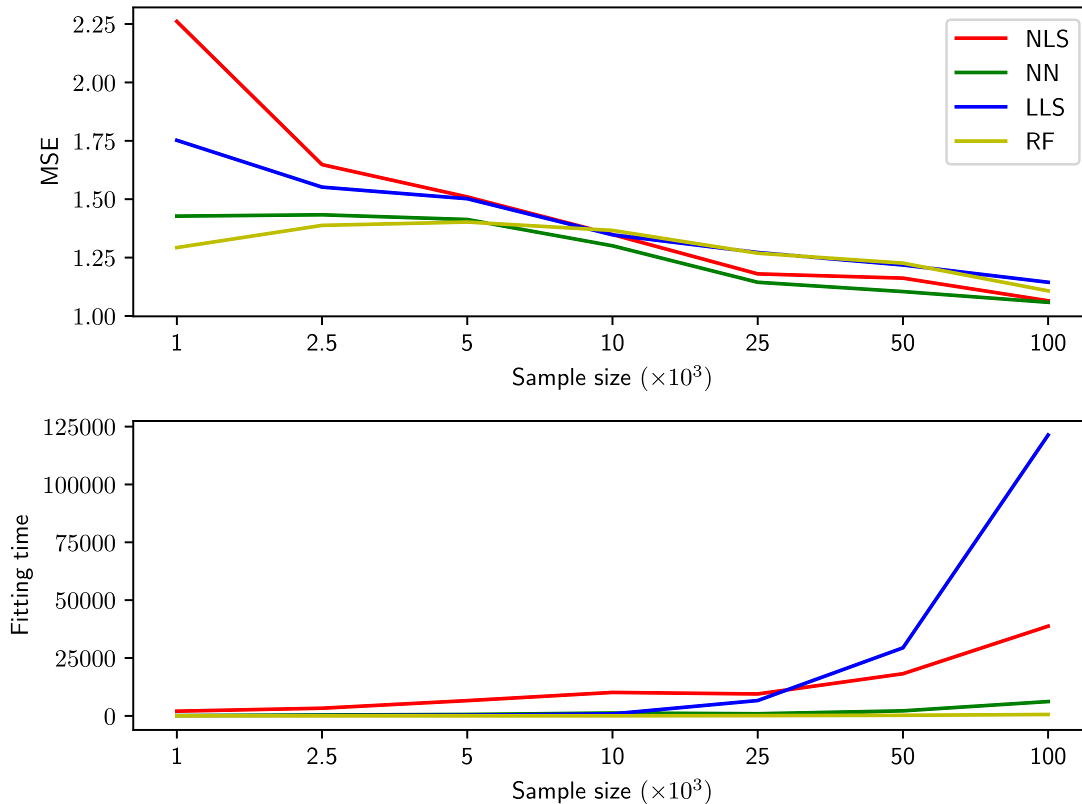

Figure 14 – Top: Models predictive accuracy over different sample sizes for the Amazon fine foods dataset. Notice that NLS performance increases in comparison to the others as the sample size grows. Bottom: Models fitting time for different sample sizes. Notice that while bigger samples massively increase the fitting time for the LLS, the NLS suffers lower impact. Both methods are slow when compared to neural network regression and random forests.

3.2.2 NLS interpretation

In machine learning, there are controversial ideas about what is a good prediction explanation (DOSHI-VELEZ; KIM, 2017). Among these, Ribeiro, Singh and Guestrin (2018) suggests that good explanations are the ones who allows human users to reproduce the regression function predictions for new samples with high accuracy, after analyzing a set of given predictions and their explanations. In this section we show how higher penalization λ allows users to reproduce NLS predictions. We use the Boston housing dataset as example.

In practice, to have a high interpretable NLS that still holds good predictive performance, we successively increase λ and check how validation MSE varies. To reduce fitting time, for each λ step, we initialize the network with the fitted weights for the immediate last value. We start with the NLS fitted in Section 3.2 and refit it for λ value in $[1, \infty]$. Figure 15 illustrates obtained MSE and average squared gradient for both training and

testing sets (as defined in Section 3.2). The figure illustrates the accuracy-interpretability trade-off occurs for the training data. On the other hand, in the test set, there is no big loss on increasing penalization factor λ in the interval $[0,50]$. Moreover, the fit with $\lambda = 5$ lead to the best test MSE values (3.61). An explanation for this fact is that the penalization controls $\Theta(\mathbf{x})$ variation and thus control over-fitting over the training data.

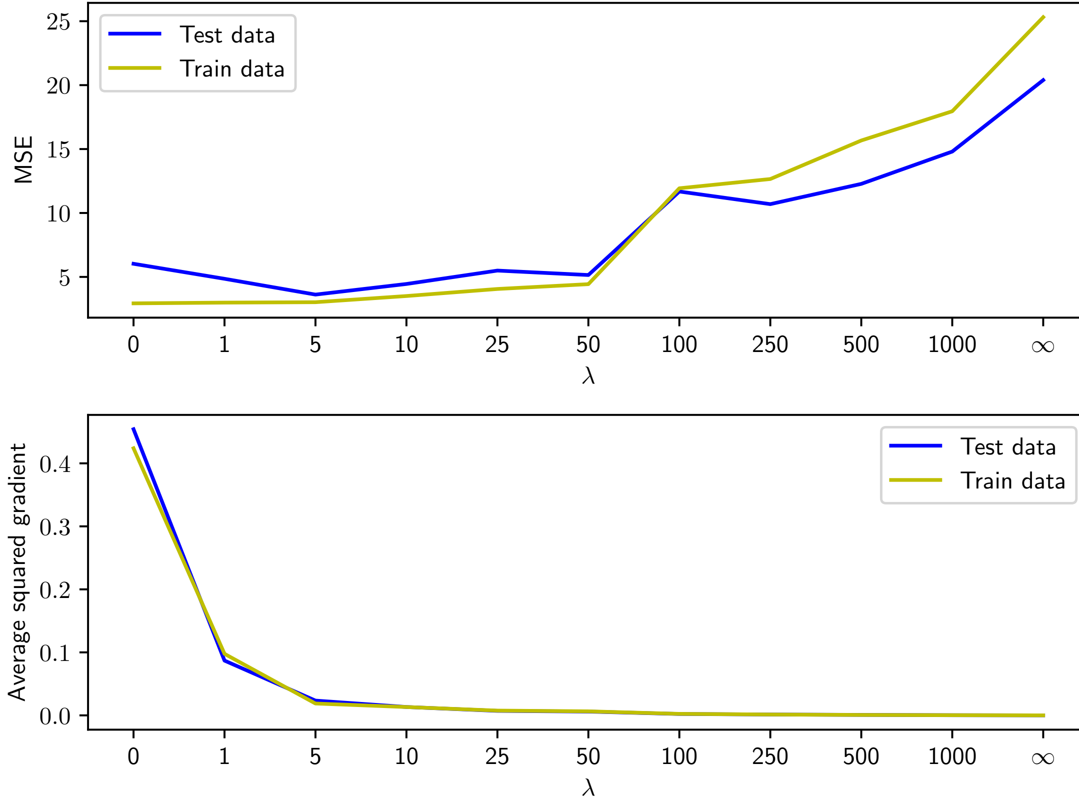

Figure 15 – MSE and average squared gradients for both training and test datasets. Higher penalization values leads to smoother $\Theta(\mathbf{x})$ estimates but higher train MSE.

As λ increases, we want to guarantee that NLS interpretations get more accurate in the sense proposed by Ribeiro, Singh and Guestrin (2018). To ensure that, we guarantee that, given seeing a set of predicted instances and theirs explanations, a naive algorithm could reproduce NLS predictions with increasingly accuracy as a function of λ. To test this statement, we propose using a set of given predictions and their explanations - for different values of λ - to obtain predictions for unseen instances though a 1 nearest neighbor approach. Algorithm 4 describe the procedure for fixed λ.

Algorithm 4 Extending interpretations to replicate predictions

Input: A set of prediction instances $\{x_1^p, ..., x_n^p\}$, a set of extension instances $\{x_1^e, ..., x_n^e\}$.
Output: Predictions obtained though interpretation extension for the extension instances.

1: **for** $x_i^e \in \{x_1^e, ..., x_n^e\}$ **do**
2: Obtain $x_{neighbor} = \arg\min_{x_j^p \in \{x_1^p, ..., x_n^p\}} d(x_i^e, x_j^p)$
3: Obtain $\theta_1(x_{neighbor}), ..., \theta_d(x_{neighbor})$ though the NLS
4: Evaluate $extended_pred(x_i^e) = \theta_0^* + \sum_{k=1}^{d} \theta_k(\mathbf{x}_{neighbor}) x_{i,k}^e$
5: **end for**

To ensure that extended predictions though interpretations are accurate, such predictions need to be compared with the ones given by the NLS. That is, we want to have low $|extended_pred(x_i^e) - true_pred(x_i^e)|$ in average. We split the test set (3/4 as prediction instances and the remaining as extension instances) and replicate Algorithm 4 for λ value in $[2, \infty]$ and obtained such averages. Figure 16 illustrates the obtained results. We conclude that the penalization strength λ indeed defines an accuracy-interpretability trade-off.

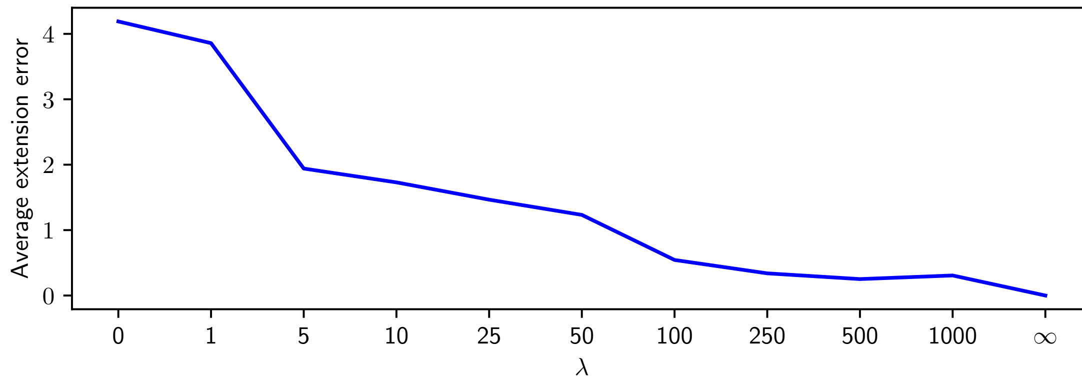

Figure 16 – Average extension error as function of the penalization strength. Notice that higher penalization values leads to more accurate prediction extensions.

CHAPTER

4

FINAL

4.1 Relationship between the NN-Stacking and the NLS

Chapter 2 described an stacking based regression estimator that linearly combine base regression estimators using non-static coefficients. Furthermore, in a UNNS, when the base estimators are the original features, that is $g_i(x) = x_i$ and the additional parameter $\phi(x)$ is used, we have,

$$G_\theta(\mathbf{x}) = \theta'_{\mathbf{x}} g_x + \phi(x) = \theta'_{\mathbf{x}} \mathbf{x} + \phi(x). \quad (4.1.1)$$

Hence, if we consider $\phi(x) = \theta_0$, this setup leads to a NLS adjustment. Although this result has no practical application, it demonstrates that these methods are straightly related.

4.2 Final Remarks and future work

4.2.1 NN-Stacking

NN-Stacking is a stacking tool with good predictive power that keeps the simplicity in interpretation of Breiman's method. The key idea of the method is to take advantage of the fact that distinct base models often perform better at different regions of the feature space, and thus it allows the weight associated to each model to vary with $\mathbf{x}$.

Our experiments show that both CNNS and UNNS can be suitable in different settings: in cases where the base estimators do not capture the complexity from the whole data, the freedom adopted by UNNS can lead to a larger improvement in performance. On the other hand, when base estimators already have high performance, UNNS the CNNS have similar predictive power, but the restrictions imposed by CNNS guarantee a more interpretable solution. Both CNNS and UNNS have comparable computational cost.

In our experiments, we observe that NNS improves over standard stacking approaches especially on large datasets. This can be explained by the fact that NNS methods have a higher complexity (i.e., larger number of parameters) than the other approaches. Thus, a larger sample size is needed to satisfactorily estimate them. The experiments also show that including weak regression methods (such as linear methods) might decrease the errors of NNS. In a few cases, however, adding such weak regressors slightly increases the prediction errors of the stacked estimators This suggests that adding a penalization to the loss function that encourages θ_i's to be zero may lead to improved results.

Future work includes extending these ideas to classification problems, as well as developing a leave-one-out version based on super learners (LAAN; POLLEY; HUBBARD, 2007). Also, we desire to develop a method of regularization on population moments estimation to avoid over-fitting, as well as to study asymptotic properties for the estimator of $L_{\mathbf{x}}$.

4.2.2 NLS

The NLS is a regression technique that applies a local linear shape to a neural network. While the NLS keeps the representation properties of the usual predictive networks, it allows users to make accurate interpretations with no need of a dedicated interpreter. NLS presents some advantages when compared to local linear smoothers as they are more consistent to irrelevant features and generate predictions faster.

We plan to further extend NLS to classification problems. Also, we plan to create experiments that directly compare interpretations given by NLS with those made by LIME.

BIBLIOGRAPHY

ALI, S.; SMITH-MILES, K. A. A meta-learning approach to automatic kernel selection for support vector machines. **Neurocomputing**, Elsevier, v. 70, n. 1-3, p. 173–186, 2006. Citation on page 50.

ARGYRIOU, A.; HAUSER, R.; MICCHELLI, C. A.; PONTIL, M. A dc-programming algorithm for kernel selection. In: ACM. **Proceedings of the 23rd international conference on Machine learning**. [S.l.], 2006. p. 41–48. Citation on page 50.

BOTARI, T.; IZBICKI, R.; CARVALHO, A. C. P. L. F. de. **Local Interpretation Methods to Machine Learning Using the Domain of the Feature Space**. 2019. Citation on page 45.

BREIMAN, L. Stacked regressions. **Machine learning**, Springer, v. 24, n. 1, p. 49–64, 1996. Citations on pages 29, 30 e 31.

____. Random forests. **Machine learning**, Springer, v. 45, n. 1, p. 5–32, 2001. Citations on pages 30 e 52.

BUZA, K. Feedback prediction for blogs. In: **Data analysis, machine learning and knowledge discovery**. [S.l.]: Springer, 2014. p. 145–152. Citations on pages 38 e 52.

CLEVERT, D.-A.; UNTERTHINER, T.; HOCHREITER, S. **Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)**. 2015. Citations on pages 37 e 48.

CSÁJI, B. C. Approximation with artificial neural networks. **Faculty of Sciences, Etvs Lornd University, Hungary**, Citeseer, v. 24, p. 48, 2001. Citation on page 36.

CYBENKO, G. Approximations by superpositions of a sigmoidal function. **Mathematics of Control, Signals and Systems**, v. 2, p. 183–192, 1989. Citation on page 46.

DAYHOFF, J. E.; DELEO, J. M. Artificial neural networks. **Cancer**, Wiley Online Library, v. 91, n. S8, p. 1615–1635, 2001. Citation on page 23.

DHEERU, D.; TANISKIDOU, E. K. **UCI Machine Learning Repository**. 2017. Available: <http://archive.ics.uci.edu/ml>. Citation on page 38.

DIETTERICH, T. G. Ensemble methods in machine learning. In: SPRINGER. **International workshop on multiple classifier systems**. [S.l.], 2000. p. 1–15. Citation on page 29.

DOSHI-VELEZ, F.; KIM, B. Towards a rigorous science of interpretable machine learning. **arXiv preprint arXiv:1702.08608**, 2017. Citations on pages 45 e 54.

DŽEROSKI, S.; ŽENKO, B. Is combining classifiers with stacking better than selecting the best one? **Machine learning**, Springer, v. 54, n. 3, p. 255–273, 2004. Citation on page 29.

FAN, J. Design-adaptive nonparametric regression. **Journal of the American statistical Association**, Taylor & Francis Group, v. 87, n. 420, p. 998–1004, 1992. Citation on page 50.

____. **Local polynomial modelling and its applications: monographs on statistics and applied probability 66**. [S.l.]: Routledge, 2018. Citation on page 50.

FAN, J.; GIJBELS, I. Variable bandwidth and local linear regression smoothers. **The Annals of Statistics**, JSTOR, p. 2008–2036, 1992. Citations on pages 30 e 50.

FRIEDMAN, J.; HASTIE, T.; TIBSHIRANI, R. **The elements of statistical learning**. [S.l.]: Springer series in statistics New York, NY, USA:, 2001. Citation on page 38.

GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. **Journal of Machine Learning Research - Proceedings Track**, v. 9, p. 249–256, 01 2010. Citations on pages 26, 37 e 48.

GOLDBERG, Y.; LEVY, O. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. **arXiv preprint arXiv:1402.3722**, 2014. Citation on page 27.

HAMIDIEH, K. A data-driven statistical model for predicting the critical temperature of a superconductor. **arXiv preprint arXiv:1803.10260**, 2018. Citations on pages 38 e 52.

HASTIE, T.; LOADER, C. *et al.* Local regression: Automatic kernel carpentry. **Statistical Science**, Institute of Mathematical Statistics, v. 8, n. 2, p. 120–129, 1993. Citation on page 50.

HECHTLINGER, Y. Interpretation of prediction models using the input gradient. **arXiv preprint arXiv:1611.07634**, 2016. Citation on page 45.

HINTON, G. E.; SRIVASTAVA, N.; KRIZHEVSKY, A.; SUTSKEVER, I.; SALAKHUTDINOV, R. R. Improving neural networks by preventing co-adaptation of feature detectors. **CoRR**, 2012. Citations on pages 26, 37 e 49.

HOERL, A. E.; KENNARD, R. W. Ridge regression: Biased estimation for nonorthogonal problems. **Technometrics**, Taylor & Francis Group, v. 12, n. 1, p. 55–67, 1970. Citation on page 48.

HORNIK, K. Approximation capabilities of multilayer feedforward networks. **Neural networks**, Elsevier, v. 4, n. 2, p. 251–257, 1991. Citation on page 51.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: BACH, F.; BLEI, D. (Ed.). **Proceedings of the 32nd International Conference on Machine Learning**. Lille, France: PMLR, 2015. (Proceedings of Machine Learning Research, v. 37), p. 448–456. Available: <http://proceedings.mlr.press/v37/ioffe15.html>. Citations on pages 26, 37 e 49.

JR, D. H.; RUBINFELD, D. L. Hedonic housing prices and the demand for clean air. **Journal of environmental economics and management**, Elsevier, v. 5, n. 1, p. 81–102, 1978. Citation on page 52.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. **Journal of artificial intelligence research**, v. 4, p. 237–285, 1996. Citation on page 27.

KHAN, J.; WEI, J. S.; RINGNER, M.; SAAL, L. H.; LADANYI, M.; WESTERMANN, F.; BERTHOLD, F.; SCHWAB, M.; ANTONESCU, C. R.; PETERSON, C. *et al.* Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks. **Nature medicine**, Nature Publishing Group, v. 7, n. 6, p. 673–679, 2001. Citation on page 23.

KHEMCHANDANI, R.; CHANDRA, S. *et al.* Optimal kernel selection in twin support vector machines. **Optimization Letters**, Springer, v. 3, n. 1, p. 77–88, 2009. Citation on page 50.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **CoRR**, abs/1412.6980, 2014. Citations on pages 26, 37 e 48.

LAAN, M. J. Van der; POLLEY, E. C.; HUBBARD, A. E. Super learner. **Statistical applications in genetics and molecular biology**, v. 6, n. 1, 2007. Citation on page 58.

MCAULEY, J. J.; LESKOVEC, J. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In: ACM. **Proceedings of the 22nd international conference on World Wide Web**. [S.l.], 2013. p. 897–908. Citation on page 52.

MCMILLEN, D. P. **Geographically weighted regression: the analysis of spatially varying relationships**. [S.l.]: Oxford University Press, 2004. Citation on page 50.

MEIR, R.; RÄTSCH, G. An introduction to boosting and leveraging. In: **Advanced lectures on machine learning**. [S.l.]: Springer, 2003. p. 118–183. Citation on page 38.

NIELSEN, M. A. **Neural networks and deep learning**. [S.l.]: Determination press USA, 2015. Citation on page 25.

NUGTEREN, C.; CODREANU, V. Cltune: A generic auto-tuner for opencl kernels. In: IEEE. **2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip**. [S.l.], 2015. p. 195–202. Citation on page 38.

PASZKE, A.; GROSS, S.; CHINTALA, S.; CHANAN, G.; YANG, E.; DEVITO, Z.; LIN, Z.; DESMAISON, A.; ANTIGA, L.; LERER, A. **Automatic differentiation in PyTorch**. 2017. Citation on page 37.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011. Citations on pages 38 e 52.

PLUMB, G.; AL-SHEDIVAT, M.; XING, E.; TALWALKAR, A. Regularizing black-box models for improved interpretability. **arXiv preprint arXiv:1902.06787**, 2019. Citation on page 48.

RIBEIRO, M. T.; SINGH, S.; GUESTRIN, C. Why should i trust you?: Explaining the predictions of any classifier. In: ACM. **Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining**. [S.l.], 2016. p. 1135–1144. Citation on page 45.

____. Anchors: High-precision model-agnostic explanations. In: **Thirty-Second AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2018. Citations on pages 47, 54 e 55.

ROJAS, R. The backpropagation algorithm. In: **Neural networks**. [S.l.]: Springer, 1996. p. 149–182. Citation on page 25.

RUAN, D.; FESSLER, J. A.; BALTER, J. Real-time prediction of respiratory motion based on local regression methods. **Physics in Medicine & Biology**, IOP Publishing, v. 52, n. 23, p. 7137, 2007. Citation on page 50.

SILL, J.; TAKÁCS, G.; MACKEY, L.; LIN, D. Feature-weighted linear stacking. **arXiv preprint arXiv:0911.0460**, 2009. Citations on pages 29 e 36.

TETKO, I. V.; LIVINGSTONE, D. J.; LUIK, A. I. Neural network studies. 1. comparison of overfitting and overtraining. **Journal of chemical information and computer sciences**, ACS Publications, v. 35, n. 5, p. 826–833, 1995. Citation on page 26.

TIBSHIRANI, R. Regression shrinkage and selection via the lasso. **Journal of the Royal Statistical Society. Series B (Methodological)**, JSTOR, p. 267–288, 1996. Citation on page 48.

VACH, W.; ROßNER, R.; SCHUMACHER, M. **Neural Networks and Logistic Regression**. 1996. Citation on page 45.

ZHANG, G.; PATUWO, B. E.; HU, M. Y. Forecasting with artificial neural networks:: The state of the art. **International journal of forecasting**, Elsevier, v. 14, n. 1, p. 35–62, 1998. Citation on page 23.

ZHOU, Z.-H. **Ensemble methods: foundations and algorithms**. [S.l.]: CRC press, 2012. Citation on page 29.