Sequential Training of Neural Networks with Gradient Boosting

Gonzalo Martínez-Muñoz

Escuela Politéctica Superior, Universidad Autónoma de Madrid, Spain

Abstract

This paper presents a novel technique based on gradient boosting to train a shallow neural network (NN). Gradient boosting is an additive expansion algorithm in which a series of models are trained sequentially to approximate a given function. A one hidden layer neural network can also be seen as an additive model where the scalar product of the responses of the hidden layer and its weights provide the final output of the network. Instead of training the network as a whole, the proposed algorithm trains the network sequentially in T steps. First, the bias term of the network is initialized with a constant approximation that minimizes the average loss of the data. Then, at each step, a portion of the network, composed of K neurons, is trained to approximate the pseudo-residuals on the training data computed from the previous iteration. Finally, the T partial models and bias are integrated as a single NN with $T \times K$ neurons in the hidden layer. We show that the proposed algorithm is more robust to overfitting than a standard neural network with respect to the number of neurons of the last hidden layer. Furthermore, we show that the proposed method design permits to reduce the number of neurons to be used without a significant reduction of its generalization ability. This permits to adapt the model to different classification speed requirements on the fly. Extensive experiments in classification and regression tasks, as well as in combination with a deep convolutional neural network, are carried out showing a better generalization performance than a standard neural network.

1 Introduction

Machine learning is becoming a fundamental piece for the success of more and more applications every day. Some examples of novel applications include bioactive molecule prediction [1], renewable energy prediction [20] or classification of galactic sources [16]. It is of capital importance to find algorithms that can handle efficiently complex data. In this context, deep architectures have shown outstanding performances specially with structured data such as images, speech, etc. [14, 19].

In addition, ensemble methods are also very effective in improving the generalization accuracy of multiple simple models [8, 5] or even complex models such as deepCNNs [17]. In recent years, gradient boosting [10], a fairly old technique, has

gained much attention by means of the computationally efficient version called eXtreme Gradient Boosting or XGBoost [6]. Apart from the gains in training speed that XGBoost achieves with respect to standard gradient boosting, XGBoost also includes a novel loss function that includes a term that controls the complexity of the trees. Furthermore, it includes several stochastic techniques to train the base models of the ensemble that has shown to improve the accuracy of the combined model, such as the ones included in stochastic gradient boosting [11] or random forest [4]. This combination of randomization techniques and optimization has placed XGBoost among the top contenders in Kaggle competitions [6] and provides very good performance in a variety of applications as in the ones mentioned above.

The objective of this study is to combine the stage-wise optimization of gradient boosting into the training procedure of a neural network. There are several related studies that propose algorithms to transform a decision forest into a single neural network [21, 2] or to use a deep architecture to train a tree forest [12, 13]. However, we could not find any proposal that applies the training procedures of ensembles to the creation of a single neural network. For instance, in [21], it is shown that a pretrained tree forest can be casted into a two-layer neural network with the same predictive outputs. First, each tree is converted into a neural network. To do so, each split in the tree is transformed into an individual neuron that is connected to a single input attribute (split attribute) and whose activation threshold is set to the split threshold. In this way, and by a proper combination of the outputs of these neurons (splits) the network mimics the behaviour of the decision tree. Finally, all neurons are combined through a second layer, which recovers the forest decision. The weights of this network can be later retrained to obtain further improvements [2]. In [12, 13], a decision forest is trained jointly by mean of a deep neural network that learns all splits of all trees of the forest. In order to guide the network to learn the splits of the trees, a procedure to train the trees using backpropagation is proposed. The final output of the algorithm is a decision forest whose performance is remarkable in image classification tasks. In this paper, we propose a combination of ensembles and neural networks that is reciprocal to [12, 13], that is, a single neural network is trained using an ensemble training algorithm.

Specifically, we propose to train the neural network iteratively as an additive expansion of simpler models. The algorithm is equivalent to gradient boosting: first a constant approximation is computed (assigned to the bias term of the neural network), then at each step a neural network with a single (or very few) neuron(s) in the hidden layer is trained to fit the residuals of the previous model. All these models are then combined to form a single neural network with one hidden layer. This training procedure provides a method that has a lower tendency to overfit as the number of combined units grow, compared to a standard neural network. In addition, it has an additive neural architecture in which the latest computed neurons contribute less to the final decision. This can be useful in computationally intensive applications as the number of active models (or neurons) can be gauged on the fly to the available computational resources without a significant loss in precision. The proposed model is tested on classification and regression problems as well as in conjunction with deep convolutional models.

The paper is organized as follows: Section 2 describes the gradient boosting and how to apply it to train a single neural network; In section 3 the results of several experimental analysis are shown; Finally, the conclusions are summarized in the last

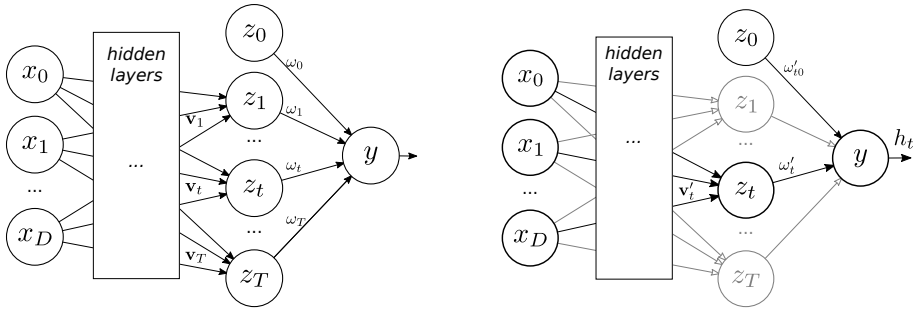

Figure 1: Illustration of a neural network and its parameters (left) and a neural network with one unit highlighted in black representing the t^{th} model trained in gradient boosted neural network. The final network built with the proposed model is as the one on the left

section.

2 Proposed method

In this section, we describe how to apply gradient boosting in the process of training a neural network as an additive model. First, we show the gradient boosting mathematical framework [10] and an extension to use this framework to any possible regression model (note that the original formulation is in some final derivations only valid for decision trees). Then, we describe how this process is integrated into a neural network. Finally, an illustrative example is given.

2.1 Gradient boosting

Given a training dataset $D = \{\mathbf{x}_i, y_i\}_1^N$, the goal of a machine learning algorithms is to find an approximation, $\hat{F}(\mathbf{x})$, of the objective function $F^*(\mathbf{x})$, which maps instances $\mathbf{x}$ to their output values y. In general, the learning process can be posed as an optimization problem in which the expected value of a given loss function, $\mathbb{E}[L(y, F(\mathbf{x}))]$, is minimized. A data based estimate can be used to approximate this expected loss: $\sum_i L(y_i, F(\mathbf{x_i}))$.

In the specific case of gradient boosting, the model is built using an additive expansion

$$F_t(\mathbf{x}) = F_{t-1}(\mathbf{x}) + \rho_t h_t(\mathbf{x}), \tag{1}$$

where ρ_t is the weight of the t^{th} function, $h_t(\mathbf{x})$. The approximation is constructed *stagewise* in the sense that at each step a new model h_t is built without modifying any of the previously created models included in $F_{t-1}(\mathbf{x})$. First, the additive model is initialized with a constant approximation such that

$$F_0(\mathbf{x}) = \underset{\alpha}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, \alpha) \tag{2}$$

and the following models are built to minimize

$$(\rho_t, h_t(\mathbf{x})) = \underset{\rho,h}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, F_{t-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i)) \quad . \tag{3}$$

However, instead of jointly solve the optimization for ρ and h_t, the problem is split into two steps. First, each model h_t is trained to learn the data-based gradient vector of the loss-function. For that, each model, h_t, is trained on a new dataset $D = \{\mathbf{x}_i, r_{ti}\}_{i=1}^{N}$, where the pseudo-residuals, r_{ti}, are the negative gradient at $F_{t-1}(\mathbf{x_i})$

$$r_{ti} = -\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\bigg|_{F(\mathbf{x})=F_{t-1}(\mathbf{x})} \tag{4}$$

The function, h_t, is expected to output values close to the pseudo-residuals at the given data points, which are parallel to the gradient of L at $F_{t-1}(\mathbf{x})$. Note, however, that the training process of h is generally guided by square-error loss, which may be different to the objective loss function. Notwithstanding, the value of ρ_t is subsequently computed by solving a line search optimization problem on the original loss function

$$\rho_t = \underset{\rho}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, F_{t-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i)) \quad . \tag{5}$$

In the following, we will focus on binary classification, in which $y \in \{-1, 1\}$, and we will use logistic loss

$$L(y, F(\cdot)) = \ln\left(1 + \exp\left(-2yF(\cdot)\right)\right), \tag{6}$$

which is optimized by the logit function $F(\mathbf{x}) = \frac{1}{2}\ln\frac{p(y=1|\mathbf{x})}{p(y=-1|\mathbf{x})}$ and similarly the constant approximation of Eq. 2 is given by

$$\begin{aligned} F_0 &= \underset{\alpha}{\operatorname{argmin}} \sum_{i=1}^{N} \ln(1 + \exp(-2y_i\alpha)) = \\ &= \frac{1}{2}\ln\frac{p(y=1)}{p(y=-1)} = \frac{1}{2}\ln\frac{1-\overline{y}}{1+\overline{y}} \end{aligned}$$

where $\overline{y}$ is the mean value of the class labels y_i. The pseuso-residuals given by Eq.4 on which the model h_t is trained for the logistic loss can be calculated as

$$r_{ti} = 2y_i/(1 + \exp\left(2y_i F_{t-1}(\mathbf{x}_i)\right)) \tag{7}$$

Once the h_t is built, the value of ρ_t is computed using Eq. 5 by minimizing

$$f(\rho) = \sum_{i=1}^{N} log(1 - exp(-2y_i(F_{t-1}(\mathbf{x}_i) + \rho h_t(\mathbf{x}_i)))) \quad .$$

There is no close form solution for this equation. However, the value of ρ can be approximated by a single Newton-Raphson step

$$\rho_t \approx -\frac{f'(\rho=0)}{f''(\rho=0)} = \frac{\sum_{i=1}^{N} r_{ti} h_t(\mathbf{x}_i)}{\sum_{i=1}^{N} r_{ti}(2y_i - r_{ti}) h_t^2(\mathbf{x}_i)} \tag{8}$$

This equation is valid for any base regressor used as additive model and not only for decision trees as in the general gradient boosting framework [10]. This analysis could be easily extended to multi-class tasks using cross-entropy loss function. For regression tasks, in which squared loss is used, this last step is unnecessary.

Finally, the output of gradient boosting composed of T models for instance $\mathbf{x}$ is given by the probability of $y = 1|\mathbf{x}$

$$p(y = 1|\mathbf{x}) = 1/(1 + \exp(-2F_T(\mathbf{x}))) \tag{9}$$

2.2 Neural network as an additive model

A multi-layered neural network can be seen as an additive model of their last hidden layer. Using the parametrization shown in Fig 1 (left), the output of the hidden last layer for a fully connected neural network for a binary tasks is

$$p(y = 1|\mathbf{x}) = \sigma\left(\sum_{t=0}^{T} \omega_t z_t\right) \tag{10}$$

with z_t being the outputs of the last hidden layer, ω_t the weights for $t = 0 \dots T$ and σ being the activation function (for classification). For regression no activation function is used. In the standard NN training all parameters of the model (i.e. $\mathbf{v}_t$ and ω_t as show in Fig. 1) are trained jointly with back-propagation. In the proposed method, all these parameters are trained sequentially using gradient boosting. To do so, after computing F_0, a fully connected regression neural network with a single neuron in the last hidden layer is trained. This neural network is trained on the residuals given by the previous iteration as given by Eq. 7. Figure 1 (right) shows (highlighted in black) the regression neural network to be trained in iteration t corresponding to model h_t. After model t has been trained, the value of ρ_t is computed using Eq. 8. Once all T models have been trained, a neural network, as shown in Fig. 1 (left), with T units in the last hidden layer can be recovered by assigning all the weights necessary to compute all the z_t variables (i.e. $\mathbf{v}_t{}'$ in Fig. 1 right) to the corresponding weights in the final NN (i.e. $\mathbf{v}_t$ in Fig. 1 left) and the weights ω_t of the output layer to

$$\omega_0 = F_0 + \sum_{t=1}^{T} 2\rho_t \omega'_{t0}$$
$$\omega_t = 2\rho_t \omega'_t \quad t = 1, \dots, T$$

Finally, in order to recover the probability of $y = 1|\mathbf{x}$ in the output of the NN as given by Eq. 9, the activation function should be a sigmoid (i.e. $\sigma = 1/(1 + \exp(-x))$ for classification with the logistic loss given in Eq. 6.

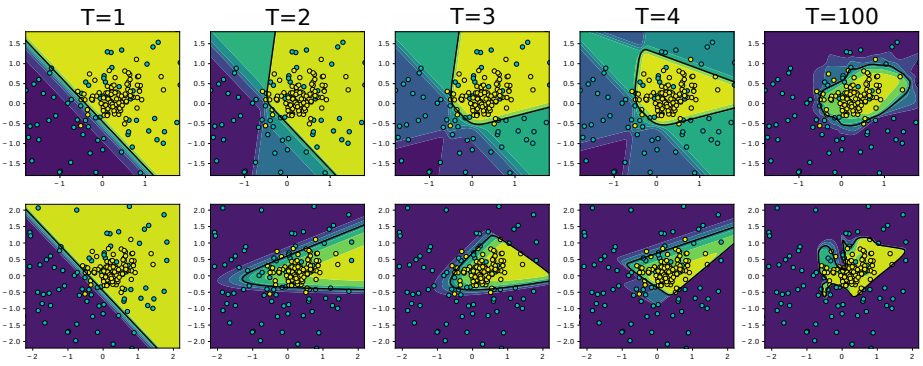

Figure 2: Classification boundaries for gradient boosted neural network (top row) and for a standard neural network (bottom row). Each column shows the results for a combination of different number of models (units). Top plots are the sequential results of a single GBNN model, whether bottom plots are independent neural networks models

This training procedure can be easily modified to larger increments of neurons, so that instead of a single neuron per step (a linear model), a more flexible model, comprising more units (K), can be trained at each iteration. The proposed training procedure can be further tuned by applying subsampling and/or shrinking, as generally used in gradient boosting [11, 9, 6]. In shrinking, the additive expansion process is regularized by multiplying each term $\rho_t h_t$ by a constant learning rate, $\nu \in (0, 1]$, in order to prevent overfitting when multiple models are combined [9]. Subsampling consists in training each model on a random subsample without replacement from the original training data. Subsampling has been shown to improve the performance of gradient boosting [11].

2.3 Illustrative example

In order to illustrate the workings of this algorithm with respect to a single hidden layer neural network, we show the performance of the method in a toy classification problem. The toy problem task consists in a 2D version of the *ringnorm* problem [3]: where both classes are 2D gaussian distribution, one with $< 0, 0 >$ mean and with covariance four times the identity matrix, and the second class with mean value at $< 2/\sqrt{2}, 2/\sqrt{2} >$ and the identity matrix as covariance.

The proposed gradient boosted neural network (GBNN) with $T = 100$ and one hidden layer is trained on 200 randomly generated instances of *ringnorm*. In addition, 100 neural nets with the number of neurons in the range $[1, 100]$ are also trained using the same training set. In Fig. 2, the boundaries for different stages of the process are shown graphically. In detail, the first and second rows show the results for GBNN and NN respectively. Each column shows the results for $T = 1$, $T = 2$, $T = 3$, $T = 4$ and $T = 100$ respectively. Note that the plots for GBNN are sequential. That is, the first column shows the first trained model; the second column, the first two models combined and so on. For the NN, each column correspond to a different NN with a

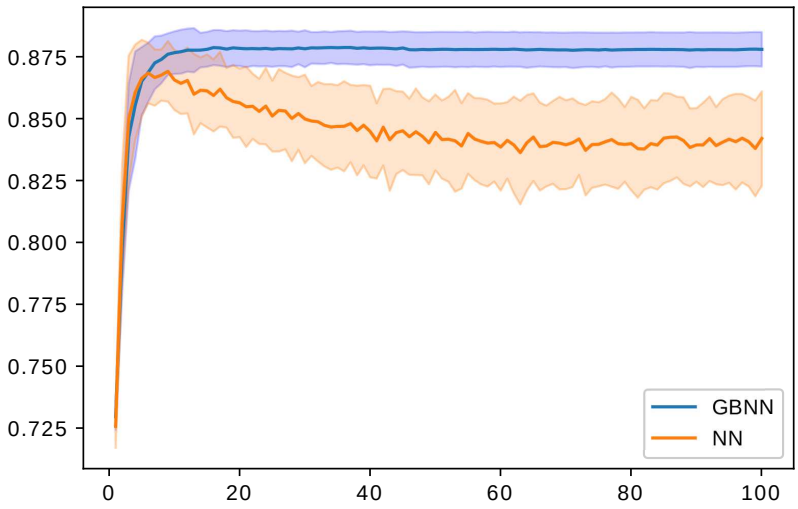

Figure 3: Average generalization accuracy of gradient boosted neural network with respect to the number of models combined (blue) and of neural networks trained with different number of hidden units (orange). The standard deviation is plotted in lighter color for each method

different number of neurons in the hidden layer as shown on top of the figure. For each column, the architecture of the networks and the number of weights (but not their values) are exactly the same. The color of the plots represent the probability $p(y = 1|\mathbf{x})$ given by the models (Eq. 9) using the *viridis* colormap. In addition, all plots show the training points.

As it can be seen from the plots both GBNN and NN start, as expected, with very similar models (column $T = 1$). As the number of models (neurons for NN) increases, GBNN builds up the boundary from previous models. On the other hand, the standard neural network, as it creates a new model for each size is able to adjust faster to the data. However, as the number of neurons increases, NN has a tendency to overfit (as shown in the bottom right most plot). On the contrary, GBNN tends to focus on the *unsolved* parts of the problem: the decision boundary becomes defined only asymptotically, as the number of models (neurons) becomes large. This characteristic is also shown in Fig. 3. That plots shows the average test error (and standard deviation in lighter color) of GBNN (in blue) and NN (in orange) with respect to the number of neurons in the last hidden layer. For this experiment, 100 random training dataset of 200 instances are created. For each training set GBNN is trained with $T = 100$ models and 100 NNs are trained with a number of hidden neurons varying up to T. Then, the performance of all models is tested on the same randomly extracted dataset composed of $10\,000$ instances. From this plot we can observe that the model trained using GBNN tend to

steadily improve the performance with respect to the models combined. However, the performance of a complex NN trained can deteriorate when larger number of units are considered (see orange line in Fig. 3).

3 Experimental results

In this section, an analysis of the efficiency of the proposed gradient boosting based neural network training is tested on twelve classification datasets coming from the UCI repository [15], six regression tasks and three applications of the proposed method that train the last hidden dense layers of a deep convolutional neural network. This last experiment was carried out in the fashion-mnist dataset [22]. These datasets have different number of instances and attributes and come from different fields of application. The `scikit-learn` package [18] was used in the experiments for neural networks. The implementation of the proposed method (gradient boosted neural network) was done in python following the standards of `scikit-learn`. The code will be made available if the manuscript is accepted. The proposed method is compared with respect to the standard neural network training procedure.

For the classification and regression experiments, single hidden layer networks were trained using the standard procedure and using the proposed gradient boosting approach. The comparison was carried out using 10-fold cross-validation and the optimum hyper-parameter setting for each method was estimated using within-train 10-fold cross validation. For the standard neural network the hyper-parameter of the number of units in the hidden layer were $[1, 3, 5, 7, 11, 12, 17, 22, 27, 32, 37, 42, 47, 52, 60, 70, 80, 90, 100, 150, 200]$. The rest of hyperparameter were set to their default values. For GBNN, a sequentially trained neural network with 200 hidden units was built in steps of K units per iteration and varying the values of the hyper-parameters as: $[0.1, 0.25, 0.5, 1]$ for the learning rate, $[0.5, 0.75, 1.0]$ for subsample rate and $K \in [1, 2, 3]$. The best sets of hyper-parameters obtained in the grid search for each method were used to train the standard neural network and the gradient boosted neural network on the whole training set. Finally, the average generalization performance (accuracy or RMSE) of the methods is estimated in the left out test set.

The deep CNN experiment was set as a transfer learning problem. For that, the dataset is first divided into a training and a test set as defined by keras library (60000 and 10000 instances for train and test respectively). Then, two of the ten classes of the problem are left out and a CNN is trained on remaining eight classes. The CNN includes the following layers: convolution(32), dropout(0.1), convolution(32), maxpooling (2x2), dropout(0.2), dense(128), dropout and an output dense layer for 8 classes. The activation functions are ReLu for the internal layers and SoftMax for the output layer. Once, this network is trained, the last layers up to dense(128) are removed and the weights of the first layers are frozen. Then, using the 12000 training instances of the left-out classes, a new dense hidden and output layers are concatenated to the frozen layers and trained. The proposed algorithm and the standard back-propagation algorithm are used to train these newly added layers. The best parametrization is obtained using within-train 10-fold cross-validation using the same parameters as above. Finally, the left-out instances of the test set are used to validate the selected model. This is a

Dataset	GBNN	NN	NN200
Classification			
Australian	85.80%±4.54	85.66%±4.20	80.74%±5.37
Banknote	100.00%±0.00	100.00%±0.00	100.00%±0.00
Breast	96.14%±1.11	96.43%±1.46	94.99%±2.41
Diabetes	74.87%±3.48	77.08%±3.39	69.79%±3.79
German	74.90%±3.45	73.90%±4.16	71.40%±4.90
Hepatitis	61.92%±13.99	60.00%±10.83	58.58%±11.76
Ionosphere	90.57%±4.75	90.04%±4.27	88.61%±5.38
Liver	69.96%±4.34	66.02%±4.54	67.38%±4.02
Magic04	87.59%±0.42	87.61%±0.29	87.17%±0.21
Sonar	81.41%±10.04	77.55%±8.86	78.05%±8.69
Spambase	94.37%±0.85	93.24%±1.23	93.33%±0.99
Tic-tac-toe	99.37%±0.95	90.30%±3.03	90.93%±2.31
Regression			
Concrete	19.04%±5.90	26.58%±11.89	88.34%±11.03
Energy	0.61%±0.16	1.42%±0.26	13.93%±5.48
Housing	9.80%±4.43	13.77%±6.09	18.79%±9.86
Power	14.29%±1.82	17.05%±1.93	70.61%±104.13
WineQ-red	0.35%±0.04	0.40%±0.05	0.41%±0.05
WineQ-white	0.42%±0.03	0.49%±0.03	0.55%±0.03
Fashion-MNIST			
Boot vs sneaker	97.55	97.45	97.65
Pullover vs coat	93.50	93.45	92.80
Shirt vs tshirt	87.85	88.70	88.15

Table 1: Average generalization performance and standard deviation for gradient boosted neural network (GBNN), a neural networks trained using a grid search (NN) and a neural network with 100 hidden units (NN100). The performance is measured as accuracy for the classification tasks and with root mean square error for regression tasts. The best results for each dataset are highlighted in a light yellow background

single partition experiment so no standard deviation can be provided. Three transfer learning problems were analyzed by leaving three sets of two classes out. Specifically the following challenging pairs were used as left out sets: T-shirt vs shirt, boot vs sneaker and pullover vs coat.

For all analyzed datasets, the average generalization performance and standard deviations are shown in Table 1 for the proposed neural network training method with 200 units (column GBNN), and for standard neural network tuned on a grid search for the number of hidden units (column NN) and a standard neural network with 200 hidden units (column NN200). The best result for each dataset is highlighted with a light yellow background. The table is structured in three blocks depending of the problem type: classification, regression and transfer learning with CNNs. An overall comparison of these results is shown graphically in Fig. 4 using the methodology described in [7]. This plot shows the average rank for the studied methods across the analyzed

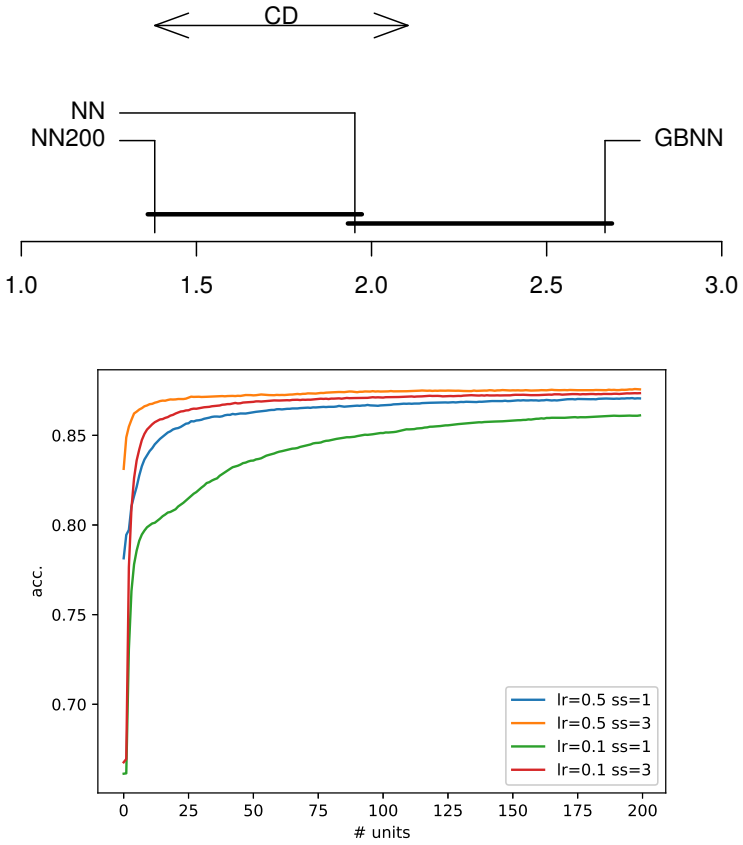

Figure 4: (top) Average ranks (higher rank is better) for GBNN, NN and NN100 (more details in the text). (bottom) Average error for several configurations of gradient boosted neural network with respect to the number of combined units ('ss' stands for the step size, K, and 'lr' for learning rate) for the magic04 classification task

datasets where higher rank indicates better results. The statistical differences between methods are determined using a Nemenyi test. In the plot, the difference in average rank of two methods is statistically significant if the methods are not connected with a horizontal solid line. The critical distance in average rank over which the performance of two methods is considered significant is shown in the plot for reference (CD = 0.72 for 3 methods and 21 datasets and p-value < 0.05).

From Table 1 and Fig. 4 (left), it can be observed that for the studied datasets, the best performing method is GBNN. This method obtains the best results in 16 out of 21 tasks. The neural net that tuned the number of units obtained the best result in five datasets and NN200 in two dataset. In classification, the differences in average accuracy among the different methods is generally favorable to GBNN. This is especially evident in *Tic-tac-toe* and *Sonar* where NN is respectively 9.1% and 3.8 worse than the result

obtained by GBNN. The most favorable result for NN is obtained in *Diabetes* where its accuracy is 2.2% better than that of GBNN. The results for NN200 are generally worse than both NN and GBNN in spite of the fact that it has the same number of hidden units of GBNN. In fact, as shown in Fig. 4 (left), the performance of NN200 is significantly worse than the performance of GBNN. In this comparison, the differences in average rank of NN are not statistically significant with either method. In regression, the results are clearly better for GBNN with respect to both NN and NN200. GBNN obtains the best results in all tested datasets.

In order to show the convergence of GBNN we will analize the *Magic04* dataset in more detail. We plot the evolution with respect to the number of hidden units for learning rates equal to $\{0.1, 0.5\}$, subsample equal to 0.5 and K equal to 1 and 3. The results of this experiment are shown in Fig. 4 (right) where the average accuracy is plotted as a function of the number of individual neurons (not models) combined. For this rather complex dataset, the plot shows a slow convergence of the curves when linear models are combined (shown as 'ss=1' in the plot). On the other hand, when more complex models are combined (composed of 3 units each, 'ss=3' in the plot) the neural network accuracy converges faster for both tested learning rates. As it can be observed from the plot, for most configurations the final trained network trained with GBNN could work combining more or less hidden units without a significant loss in performance. These curves indicate that the hidden neurons in the final trained network are *ordered* by importance. However, further analysis is necessary to explore this property in more detail.

4 Conclusions

In this paper we present a novel method to train a shallow neural network iteratively based on gradient boosting algorithm. The proposed algorithm builds at each step a regression neural network with K units that minimizes a given loss function by fitting the data residuals. After each model is built the residuals are updated to train the model of the next iteration. The resulting T models can be integrated into a single neural network with $T \times K$ hidden units. In the analyzed problems the proposed method achieves a generalization accuracy that converges asymptotically with respect to the number of combined models (neurons). Whether in standard neural networks, the generalization accuracy generally reaches a maximum at a given number of neurons and then progressively deteriorates as the number of units is increased. This quality of the proposed method allows us to use the combined model fully or partially by deactivating the units in order inverse to their creation depending on classification speed requirements.

The proposed method was tested on a variety of classification and regression tasks. The results show a performance favorable to the proposed method in general. This is specially evident for regression tasks where the proposed method achieved the best result in all the analyzed datasets.

References

[1] Ismail Babajide Mustapha and Faisal Saeed. Bioactive molecule prediction using extreme gradient boosting. *Molecules*, 21(8), 2016.

[2] Gérard Biau, Erwan Scornet, and Johannes Welbl. Neural random forests. *Sankhya A*, In press, 2018.

[3] L. Breiman. Bias, variance, and arcing classifiers. Technical Report 460, Statistics Department, University of California, 1996.

[4] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 161–168, New York, NY, USA, 2006. ACM Press.

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[7] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[8] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, 15:3133–3181, 2014.

[9] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[10] Jerome H. Friedman. Greedy function approximation: a Gradient Boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001.

[11] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367 – 378, 2002. Nonlinear Methods and Data Mining.

[12] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo. Deep neural decision forests. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[13] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo. Deep neural decision forests. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[15] M. Lichman. UCI machine learning repository, 2013.

[16] N. Mirabal, E. Charles, E. C. Ferrara, P. L. Gonthier, A. K. Harding, M. A. Sánchez-Conde, and D. J. Thompson. 3fgl demographics outside the galactic plane using supervised machine learning: Pulsar and dark matter subhalo interpretations. *The Astrophysical Journal*, 825(1):69, 2016.

[17] Mohammad Moghimi, Mohammad Saberian, Jian Yang, Li-Jia Li, Nuno Vasconcelos, and Serge Belongie. Boosted convolutional neural networks. In *British Machine Vision Conference (BMVC)*, York, UK, 2016.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] Jrgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[20] Alberto Torres-Barrán, Álvaro Alonso, and José R. Dorronsoro. Regression tree ensembles for wind energy and solar radiation prediction. *Neurocomputing (2017)*, 2017.

[21] Johannes Welbl. Casting random forests as artificial neural networks (and profiting from it). In Xiaoyi Jiang, Joachim Hornegger, and Reinhard Koch, editors, *Pattern Recognition*. Springer International Publishing, 2014.

[22] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.