Investigating latent representations and generalization in deep neural networks for tabular data

Edouard Couplet[a,*], Pierre Lambert[a], Michel Verleysen[a], John A. Lee[a,1], Cyril de Bodt[a]

[a] *UCLouvain - ICTEAM/ELEN, Place du Levant 3, bte L5.03.02, Louvain la Neuve, 1348, Belgium*
[b] *UCLouvain - IREC/MIRO, Avenue Hippocrate 55, B1.54.07, Brussels, 1200, Belgium*

Abstract

Recent deep neural network architectures that are tailored to tabular data operate at the feature level and process multiple latent representations simultaneously, typically one per feature. We investigate the impact of varying the dimension and number of such latent representations on model performance and generalization. Our results identify distinct model behaviors during both training and testing phases. To ease analysis of these behaviors, we propose a novel tool for characterizing data complexity and use it to highlight intricate relationships between data complexity, model complexity and model performance. We hypothesize a phenomenon of implicit self-regularization which intensifies with model capacity and sample-to-dimension ratio. While this self-regularization can mitigate over-fitting, it may also lead to reduced performance on training data. Our findings expand the understanding of neural networks applied to tabular data and provide insights that can help practitioners and/or automated methods in designing neural networks architectures that better match the complexity of specific tabular data sets.

Keywords:
tabular data, neural networks, generalization, deep learning, data complexity, model complexity

*Corresponding author
Email address: edouard.couplet@uclouvain.be (Edouard Couplet)

Introduction

Despite the widespread success of deep learning in handling homogeneous data types like text or images, the challenge of processing heterogeneous tabular data persists. As of now, ensemble models based on gradient boosted decision trees such as XGBoost [1], CatBoost [2], and LightGBM [3], continue to demonstrate superior performance in many scenarios involving tabular data [4], often with significantly lower computational requirements. Nevertheless, deep neural networks remain an attractive option, primarily due to their great flexibility and remarkable capabilities in representation learning. These attributes make them particularly well-suited for multi-modal tasks requiring the integration of data, including tables, from a wide variety of sources [5].

One of the main difficulties of learning on tabular data lies in the heterogeneous nature of its features and the absence of clear underlying structure that can be exploited as inductive bias. The community has therefore naturally favored neural network architectures that are geared towards modelling intricate relationships between features. In recent years in particular, with the rise and establishment of graph neural networks (GNNs) and transformers, efforts have multiplied to adapt these specialized architectures to models suitable for tabular data. Notable examples of GNN-based tabular models include Fi-GNN [6] and Table2Graph [7]. Among the first tabular models to use attention-based mechanisms was AutoInt [8], closely followed by TabNet [9] and TabTransformer [10]. Subsequently, models like FT-transformer [11], SAINT [12], and more recently T2G-Former [13] have emerged, the latter combining typical transformer-like multi-head attention with elements of graph learning.

These models all use one separate latent representation per feature: a token in case of transformers, a node vector in case of GNNs. Consequently, we classify them as *feature-level* models. In contrast, Multi-layer Perceptrons (MLPs) or ResNets use one combined latent representation per sample, and we classify them as *sample-level* models. Extensive benchmarking shows that well-tuned *sample-level* models tend to be strong baselines [14] and that no recently proposed *feature-level* model consistently outperforms all the others [15]. All in all, we still lack significant insights into what design choices render *feature-level* architectures effective or ineffective on a given tabular dataset. For example, while using one representation per feature seems an obvious choice and eases the interpretation of latent representations as contextual

embedding of the original features, this is not strictly enforced by the models: just as we can adjust the dimensions of these latent representations, there is no restriction on their number.

In a preliminary work [16], we have shown that, depending on the dataset, reducing or augmenting the number of latent representations can help improve performance. In this work, we extend our analysis to the effect of dimension. We implement a generic *feature-level* architecture and strive to understand how both dimension and number of latent representations impact model behavior and generalization capabilities. Additionally, as we expect these impacts to be highly dataset dependent, we also propose a tool for analyzing data complexity and aid in our investigation.

While perhaps less than model complexity, data complexity has long been discussed in the machine learning community, mainly for classification problems. Proposed data complexity measures are based on volume of overlapping regions, class separability, or data distribution [17]. These measures have also been extended to regression problems, e.g., in [18]. Our proposed tool follows the methodology introduced in [19] where the authors evaluate how a growing subset of the data is able to approximate the data distribution. However, instead of using a measure of the distance between two distributions, we measure the non-linearity of a model fitted on growing subsets of nearest neighbors. This is less universal, but better reflects how ReLU activated neural networks may approximate mapping from attributes to labels.

Our results suggest the existence of an implicit self-regularization effect which grows with model capacity, with particular sensitivity to the number of latent representations. While this implicit regularization effect can help mitigate over-fitting, it can also lead to severe under-fitting, especially when the data has a high complexity and/or a high ratio between the number of samples and the number of features. These findings broaden our comprehension of neural networks in the context of tabular data. They provide insights that can help identify failure cases, improve model robustness, and guide practitioners in designing optimal architectures for specific supervised learning tasks and tabular data sets. This research could also have practical implications in automated Machine Learning and related subfields like Meta-Learning, and Neural Architecture Search (NAS), where the goal is to automate/learn the learning process itself. In NAS, for example, efficient exploration of the search space is critical for finding the best architectures [20] and has been a key factor driving the field forward [21]. In this context, a better understanding of the intricate relationships between model complex-

ity and data complexity could help inform heuristics for narrowing down the search space and enhancing resource efficiency.

In Section 1, we provide background information on tabular data and present a formalization of *sample-level* models and *feature-level* models. In Section 2, we design a simple, generic *feature-level* architecture. In Section 3, we test this architecture on various datasets and examine how performance is affected by varying the dimension and number of latent representations. In Section 4, we propose a data complexity analysis tool and use it to interpret the results from Section 3. Finally, Section 5 attempts to give a broader picture and suggests avenues for future research, before drawing some conclusions in Section 6.

1. Sample-level and feature-level models for tabular data

This study focuses primarily on supervised learning tasks with tabular data, one of the most ubiquitous data modalities. Tabular data is organized into rows and columns where each row corresponds to a sample and each column to an attribute, or feature. A data set is denoted $X = \{(\mathbf{x^{(i)}}, y^{(i)})\}_{i=1}^{n}$ where $\mathbf{x^{(i)}} = (x_1^{(i)}, x_2^{(i)}, ..., x_m^{(i)}) \in \mathbb{X}$ represents the features of the ith sample and $y^{(i)} \in \mathbb{Y}$ denotes the sample label. We consider two types of tasks: binary classification when $\mathbb{Y} = \{0, 1\}$ and regression when $\mathbb{Y} = \mathbb{R}$.

A particularity of tabular data is that features are not necessarily of the same nature. Formally, we say that tabular data can exhibit heterogeneity in its feature space: for two given features $x_j \in \mathbb{X}_j$ and $x_{j'} \in \mathbb{X}_{j'}$, it is common that $\mathbb{X}_j \neq \mathbb{X}_{j'}$. This heterogeneity may include a mix of numerical and categorical features and poses a number of challenges for algorithmic processing.

Additionally, tabular data does not entail an obvious underlying structure that relates different features to one another. Rows and columns can be permuted without any impact. This is not the case for other data modalities such as images or text. In images, for example, pixels are arranged on a grid and obey some spatial structure. They cannot be interchanged without breaking this structure. In the context of deep learning, structure is exploited as an inductive bias to design specialized neural network architectures; in the case of images, convolutional neural networks. Because tabular data lacks such strong inductive bias, there are currently no specialized neural network architecture that dominates the field and a multitude of different approaches have been proposed. We classify these approaches into

two categories: *sample-level* models and *feature-level* models. They differ in how they represent data in their latent spaces. The former use one latent vector for one sample and correspond to more traditional architectures such as MLPs or ResNets. The latter use multiple latent vectors for one sample: one for each latent feature, and correspond to more recent architectures such as graph neural networks or transformers. In the next sections, we give a more formal definition for *sample-level* models and *feature-level* models.

1.1. Sample-level models

The task of label prediction is formulated as an encoding-decoding problem. We propose the following general formulation for *sample-level* models[1]:

$$\mathbf{y} = \rho_\psi(\phi_\theta(f(\mathbf{x}))) \ ,$$

where $f(\cdot)$ represents the *pre-encoder*, where $\phi_\theta(\cdot)$ is a neural network parameterized by θ and represents the *encoder*, and where $\rho_\psi(\cdot)$ is a neural network parameterized by ψ and represents the *decoder*.

Pre-encoder. Standard neural networks accept only vectors of real numbers as input. Due to the heterogeneous nature of tabular data, we require a preprocessing step to encode each feature x_j with a specific function $f_j : \mathbb{X}_j \to \mathbb{R}^{d_j}$. In *sample-level* models, all encodings $f_j(x_j)$ are typically concatenated into a single sample representation of dimension $d = \sum_j d_j$.

Encoder. The objective of the encoder is to capture intricate patterns within the data and learn an appropriate representation which enables the decoder to easily (and accurately) predict labels. It is a neural network with $l + 1$ hidden layers $\phi_k : \mathbb{R}^{d_{k-1}} \to \mathbb{R}^{d_k}$ such that $\phi = \phi_l \circ \phi_{l-1} \circ ... \circ \phi_1 \circ \phi_0$. The input layer $\phi_0 : \mathbb{R}^d \to \mathbb{R}^{d_0}$ is often referred to as the embedding layer. The output $\mathbf{z} = \phi(\mathbf{x})$ of the last layer of the encoder is passed on to the decoder.

Decoder. The decoder is another neural network $\rho : \mathbb{R}^{d_l} \to \mathbb{Y}$ with $\rho(\mathbf{z}) = \hat{y}$, where $\hat{y}$ denotes the predicted label for a given sample $\mathbf{x}$.

[1]The terminology and notations that are used here originate from the literature on deep sets and graph neural networks.

1.2. Feature-level models

Similarly to *sample-level* models, we propose the following general formulation for *feature-level* models:

$$\mathbf{y} = \rho_\psi(\text{pool}(\phi_\theta(f(\mathbf{x})))) \ ,$$

where $f(\cdot)$ represents the *pre-encoder*, where $\phi_\theta(\cdot)$ is a neural network parameterized by θ and represents the *encoder*, and where $\rho_\psi(\cdot)$ is a neural network parameterized by ψ and represents the *decoder*. Compared to the *sample-level* formulation, there is the addition of a pooling operator before the decoder.

Pre-encoder. in *feature-level* models, we project all features x_j into a common space $\mathbb{R}^d$ and maintain a separate vector representation for each feature. We define $f : \mathbb{X} \to \mathbb{R}^{d \times m}$ as $f(\mathbf{x}) = (f_1(x_1), f_2(x_2), ..., f_m(x_m))$.

Encoder. The encoder is a neural network with $l + 1$ hidden layers $\phi_k : \mathbb{R}^{d_{k-1} \times m_{k-1}} \to \mathbb{R}^{d_k \times m_k}$ such that $\phi = \phi_l \circ \phi_{l-1} \circ ... \circ \phi_1 \circ \phi_0$. The input layer $\phi_0 : \mathbb{R}^{d \times m} \to \mathbb{R}^{d_0 \times m_0}$ is often referred to as the embedding layer. The final m_l latent representations of dimension d_l learned by the encoder are combined together via a pooling operator into $\mathbf{z} \in \mathbb{R}^{d_l}$.

Decoder. As for *sample-level* models, the decoder is another neural network $\rho : \mathbb{R}^{d_l} \to \mathbb{Y}$ with $\rho(\mathbf{z}) = \hat{y}$, where $\hat{y}$ denotes the predicted label for a given sample $\mathbf{x}$.

Gorishniy et al. investigate different approaches for the pre-encoder as well as various combinations with dedicated embedding layers in the encoder [22]. The authors show that model performance can be significantly improved by selecting appropriate encoding functions f_j and embedding layers, thereby encouraging further exploration of the subject. However, in practice, the pre-encoder is usually straightforward and similar across models: f_j is either the identity or a normalizing function when x_j is a numerical feature, and f_j is a one-hot encoder when x_j is a categorical feature. Likewise, the decoder is most often composed of one or two fully connected linear layers with appropriate nonlinear activations depending on the task. What really sets apart recent *feature level* deep learning models for tabular data is the design of the encoder. Fi-GNN [6] uses a form of graph attention network with a

fully connected feature graph; Table2Graph [7] relies on reinforcement learning and edge regularization to learn a sparse feature graph. AutoInt [8] uses standard attention heads with residual connections; TabNet [9] relies on a sequential multi-step attention process and aggregates results from each step to produce final output, TabTransformer exploits multi-head attention[10]. FT-transformer [11] uses features biases in the embedding layers and relies on a backbone which is much more similar to vanilla transformers [23] than previous models, SAINT [12] incorporates inter-sample attention, and T2G-Former [13] also uses typical transformer blocks but computes attention coefficients based on a feature graph. A shared aspect in all these encoder designs, however, is that the number m_k of latent representation in each hidden layer is always equal to m, the number of features in the original data (or $m+1$ in case a CLS output token is added, as in FT-transformer). While this setup enables a clear interpretation of latent representations as contextual feature embeddings, it raises the question of whether m_k must inevitably remain fixed at m and whether, together with dimension d_k, it could be modified to improve model performance.

2. A simple feature-level architecture

To conduct our investigation, we design a simple and versatile *feature-level* architecture for tabular data. Figure 1 shows a schematic representation of this architecture.

Pre-encoder. We define the *pre-encoder* feature-wise; for a given feature j, we have:

$$f_j(\cdot) = \text{pad}\left(g_j(\cdot)\right), \text{ with } g_j(\cdot) = \begin{cases} \text{gaussianize}(\cdot) & \text{if } j \text{ is numerical} \\ \text{one-hot-encode}(\cdot) & \text{if } j \text{ is categorical} \end{cases}.$$

Here, gaussianize$(\cdot)$ applies a quantile transformation to ensure the jth feature conforms to a normal distribution, one-hot-encode$(\cdot)$ converts its input into a binary vector, and pad$(\cdot)$ supplements zeroes to the resulting encoding such that $\sum_j f_j(x_j) = \text{concatenate}_j(g_j(x_j))^T$ for a given sample $\mathbf{x}$. This ensures that the m initial feature encodings f_j are living in the same space $\mathbb{R}^d$ but stay independent of one another. Putting everything together, we obtain sample pre-encoding $\mathbf{e} = f(\mathbf{x})$, with $\mathbf{e} \in \mathbb{R}^{d \times m}$.

Encoder. The *encoder* ϕ is composed of two hidden layers: ϕ_0 and ϕ_1. The first layer ϕ_0 takes as input $\mathbf{e} \in \mathbb{R}^{d \times m}$ and outputs $\mathbf{h}_0 = \phi_0(\mathbf{e}) \in \mathbb{R}^{d' \times m'}$. The second layer ϕ_1 takes as input $\mathbf{h}_0 \in \mathbb{R}^{d' \times m'}$ and outputs $\mathbf{h}_1 = \phi_1(\mathbf{h}_0) \in \mathbb{R}^{d' \times m'}$. The dimension d' and number m' of latent representations are not necessarily equal to the dimension d and number m of pre-encoded original features. The equations for the two layers are:

$$\mathbf{h_0} = \phi_0(\mathbf{e}) \;\; = \mathrm{ReLU}(W_0\tilde{\mathbf{e}} \;\; + \mathbf{b}_0\mathbf{1}^T) \;\; \text{with} \;\; \tilde{\mathbf{e}} \;\; = \mathbf{e}A_0 \;\; , \tag{1}$$

$$\mathbf{h_1} = \phi_1(\mathbf{h}_0) = \mathrm{ReLU}(W_1\tilde{\mathbf{h}}_0 + \mathbf{b}_1\mathbf{1}^T) \;\; \text{with} \;\; \tilde{\mathbf{h}}_0 = \mathbf{h}_0A_1 \;\; , \tag{2}$$

where W_k, $\mathbf{b}_k$ and A_k are trainable parameters and $\mathbf{1}$ is a vector of d' ones. The pooling operator is simply a sum: $\mathbf{z} = \sum_{j=1}^{m'} h_{1j}$.

Decoder. The *decoder* takes $\mathbf{z}$ as input and produces a label prediction $\hat{y}$. It comprises a single linear layer: $\hat{y} = W\mathbf{z} + b$. In the case of regression tasks, we minimize the mean square error. Meanwhile, for classification tasks, we incorporate a sigmoid activation function and minimize the binary cross entropy.

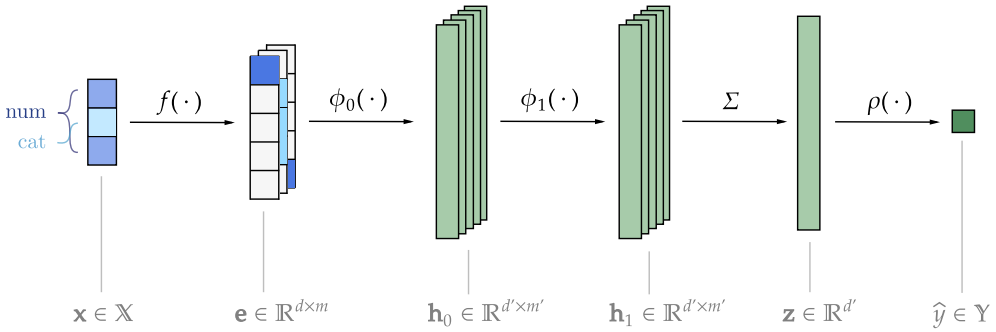


Figure 1: Generic *feature-level* architecture for tabular data. Equations for computing $\mathbf{e}, \mathbf{h}_0, \mathbf{h}_1, \mathbf{z}$, as well as the final prediction $\hat{y}$ are detailed in the main text of Section 2

.

This model is *generic* in the sense that one can easily recover other neural network architectures. If A_k are square matrices of size $m \times m$ in which columns are constrained to sum up to one, we obtain attention matrices and a simplified version of a transformer (without residual connections and where the attention coefficients are learned directly). Simple graph convolutional

networks would correspond to the case where A_k is the adjacency matrix of the computational graph. If A_0 is a diagonal matrix, ϕ_0 can be interpreted as a static embedding layer (e.g., similar to the first layer of FT-transformer and other variants). If A_0 is a vector of ones and A_1 a simple scalar one, we recover standard MLPs.

3. Evaluation of model performance

In this section, we evaluate the performance of our generic architecture on 12 different datasets and discuss how it is affected by the choice of m' and d', the number and dimension of latent representations. The datasets come from the benchmark introduced in [4]. We consider both regression and classification tasks, as well as datasets with both categorical and numerical features. Table 1 summarizes the main attributes of selected datasets.

Name	Task	n samples	m_{cat}	m_{num}	m	d
Bank-marketing	Classification	10578	0	7	7	7
Bike_Sharing_Demand	Regression	17379	5	6	11	20
California	Regression	20640	0	8	8	8
Compass	Classification	16644	9	8	17	59
Electricity	Classification	38474	1	7	8	14
Eye_movements	Classification	7608	3	20	23	26
House_sales	Regression	21613	2	15	17	19
KDDCup09_upselling	Classification	5032	11	34	45	85
Rl	Classification	4970	7	5	12	37
Sulfur	Regression	10081	0	6	6	6
Superconduct	Regression	21263	0	79	79	79
Wine_quality	Regression	6497	0	11	11	11

Table 1: Datasets characteristics, with m being the number of original features, m_{cat} the number of categorical features, m_{num} the number of numerical features, and d the dimension of the feature representation after the pre-encoding stage.

3.1. Experiment settings

In the *pre-encoder*, we use ScikitLearn's `OneHotEncoder` for categorical features and its `QuantileTransformer` for numerical features. We test the following values for $m' : \{1, 2, 4, 8, 16, 32, 64, 128\} \cup \{m\}$ and $d' : \{16, 32, 64, 128, 256\}$.

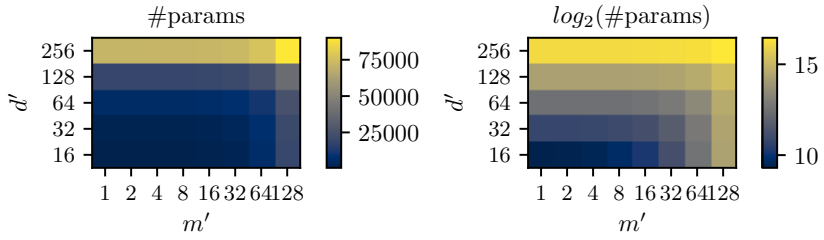


Figure 2: Trainable parameter count for our generic *feature-level* model. We consider a case where $m = d = 20$.

Figure 2 shows heat maps that indicate the magnitude of the number of trainable parameters for all possible combinations of m' and d'.

Data gets split into training and test sets (70% and 30%, respectively, and 30% of the training data serves as a validation set for monitoring). The batch size is 256 and each model learns for 200 epochs with an Adam optimizer and a learning rate of 0.001. These operations get repeated for each dataset starting from 10 different random seeds.

The averaged test results are reported in Fig. 3. For classification tasks, we measure the test accuracy; for regression tasks, we measure the test R^2 score.

3.2. Results

We focus our analysis on four prototypical datasets that are representative of the four different kinds of behavior that can be observed:

Case a. `superconduct` - In this scenario, increasing the number of parameters consistently improves training performance. Although some degree of over-fitting is present, no decrease in test performance occurs as the parameter count grows.

Case b. `bank marketing` - Having more parameters improves performance on training data but not on test data: obvious over-fitting is observed. Depending on the value of dimension d', a lower or higher number m' of latent representations happens to be better.

Case c. `sulfur` - Surprisingly, the model with the highest number of parameters is not the one that performs best on the training data. We observe no over-fitting, the best combination is a high d' but a low m'.

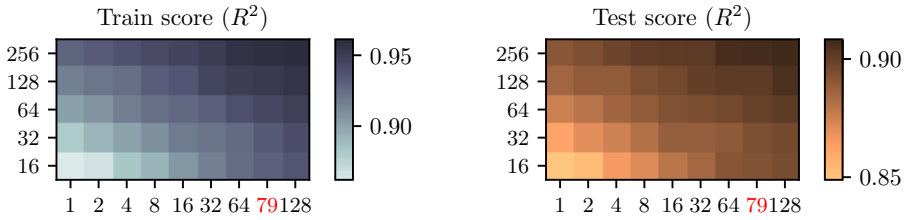
Case d. `california` - Similarly, a high d' together with a low m' lead to the best performance on the training data. In this case, though, the model also over-fits such that the best performance on the test set actually occurs with moderate values of both d' and m'.

4. Analysis of model behavior and generalization

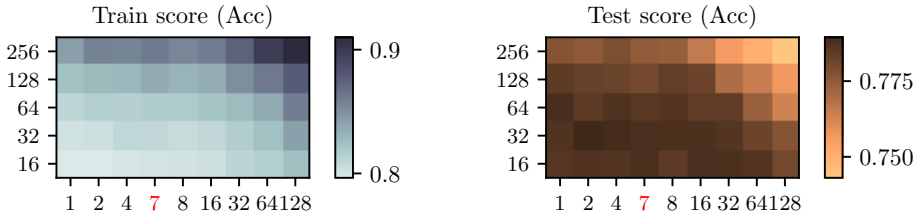
In order to shed light on our results, we involve notions of model complexity and data complexity, which will be defined and developed further below. At this point, however, let us already sketch two important distinctions: *model expressive complexity* versus *model effective complexity*, and *data observed complexity* versus *data intrinsic complexity*. Model expressive complexity, also referred to as expressive power, measures the capacity of a model to learn complex functions 'in theory', while effective complexity reflects the complexity of the function that is learned in practice by the model [24]. The observed data complexity denotes the complexity of the data as it is observed and measured, with accompanying noise, whereas intrinsic data complexity refers to the complexity of the underlying noiseless data generative function. The results that are reported in the previous section raise two related questions that we can formulate in terms of model complexity and data complexity:

Question 1. How does the choice of the dimension d' and number m' of latent representations influence the performance on the training data (cases a and b versus cases c and d)? Since we observe different model behaviors on different datasets with same values for d' and m', we assume that the answer to this question is dependent on the observed complexity of data. Moreover, the value of d' and m' will control the model expressive complexity, while the performance on the training data is closely related to the model effective complexity. Investigating this question is thus trying to understand how effective model complexity is related to model expressive complexity and data observed complexity.

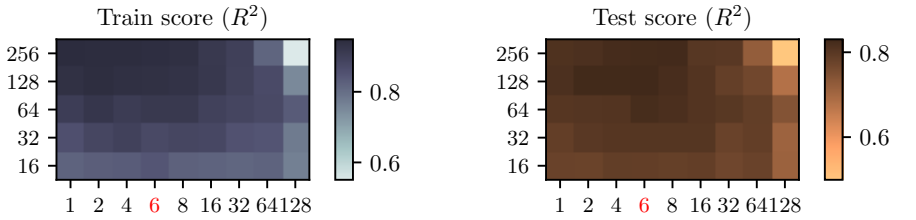
Question 2. How does the choice of d' and m' influence the performance on the test data (cases a and c versus cases b and d)? This is essentially asking about the generalization of the model and whether it will over-fit or not. We can describe over-fitting as a situation where model effective complexity is close to the data observed complexity, but where a significant gap separates data observed complexity and data intrinsic complexity, causing an



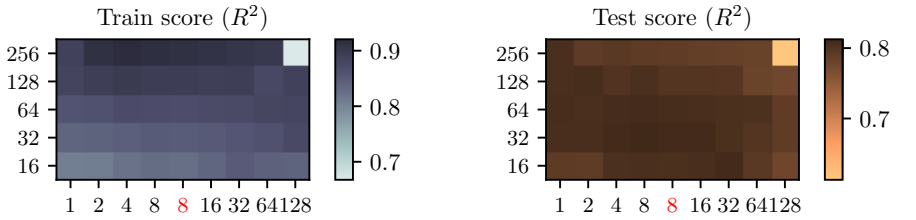
(a) Scores for `superconduct` (regress.)



(b) Scores for `bank-marketing` (classif.)



(c) Scores for `sulfur` (regress.)



(d) Scores for `california` (regress.)

Figure 3: Results for 4 different dataset. Performance on the training set is reported in the left column. Performance on the test set is reported in the right column. Performances are averaged over 10 random folds. In each heat map, the vertical axis represents the possible values for the dimension d' of the latent representations, while the horizontal axis indicates the possible values for the number m' of latent representations. The value that is highlighted in red corresponds to the standard case where $m' = m$.

important generalization error. Since this question relies on effective model complexity, it logically follows from the first one.

Both questions also heavily relate to the concept of data complexity, warranting the initial characterization of such complexity. First, we propose a tool for this purpose. Next, leveraging this tool, we delve into the successive investigation of *question 1* and *question 2*.

4.1. A tool for measuring data complexity

For a dataset $X = \{(\mathbf{x^{(i)}}, y^{(i)})\}_{i=1}^{n}$ where $\mathbf{x^{(i)}} = (x_1^{(i)}, x_2^{(i)}, ..., x_m^{(i)}) \in \mathbb{X}$ and $y^{(i)} \in \mathbb{Y}$, we define observed data complexity as the non-linearity of the discrete mapping $\mathbb{X} \to \mathbb{Y}$. As an example, if the points $\{(\mathbf{x^{(i)}}, y^{(i)})\}_{i=1}^{n}$ lie on a hyperplane (regression) or are linearly separable (classification), the observed data complexity would be zero. If instead the points are distributed randomly, the complexity would be maximal. A first solution for computing this complexity would be to fit a linear model to the data and take the error as a measure of the observed complexity. However, this would only tell us if the data is linear or non-linear on a global scale. For a more refined analysis, a multi-scale view of the data is preferred. To achieve this, we fit a local linear model within a neighborhood of size k around each point $\mathbf{x}^{(i)}$ and make a prediction $\hat{y}^{(i)}$ for that same point with the fitted model (we use a standard regressor for regression problems, and a logistic regressor for classification problems). We obtain a 'smoothed' approximation $\hat{\mathbf{y}}$ of the data. We then compute and aggregate the errors between the approximation $\hat{\mathbf{y}}$ and the actual labels $\mathbf{y}$ using an appropriate scoring metric $S(\hat{\mathbf{y}}, \mathbf{y})$ (we use R^2 for regression and Accuracy for classification, so a value of 1 would mean perfect linearity for scale k). By varying the size k of the neighborhood, we can obtain either coarser or finer approximations and sketch a complete 'linearity profile' of the data. A more detailed procedure is provided in Algorithm 1, while Fig. 4 illustrates the procedure executed on a toy regression dataset.

Algorithm 1: Linearity profiler

input : Dataset $\{(\mathbf{x^{(i)}}, y^{(i)})\}_{i=1}^{n}$
Neighborhood sizes $\{k^{(p)}\}_{p=1}^{q}$
output: Linearity profile $\{(r^{(p)}, s^{(p)})\}_{p=1}^{q}$

```
% Initialization
S(·) ← Scoring function % R² if regression, Accuracy if classification
linearity_profile ← list() % empty list of size q
for k^(p) in Neighborhood sizes do
    ŷ ← list() % empty list of size n for storing local predictions
    for (x^(i), y^(i)) in Dataset do
        L(·) ← Linear model
        ind ← indices of the k^(p) nearest neighbor of x^(i)
        fit L on {(x^(j), y^(j))}_{j ∈ (ind ∪ i)}
        ŷ[i] ← L(x^(i)) % make local prediction for point i
    end
    r ← n/k^(p) % resolution
    s ← S(y, ŷ) % deviation from linearity at resolution r
    linearity_profile[p] ← (r, s)
end
```

In the end, we get a curve of either R^2 or Accuracy with respect to resolution, which is defined as n/k. Resolution can be loosely interpreted as the number of independent linear models that are fitted to the data, although, in fact, we always fit one model per point, but if k is large, these n models will strongly depend on each other. As an example, when $k = n$ (resolution=1), we fit a linear model at each point using every other points: the n models are thus exactly the same. Moreover, if we obtain a score of 1 at resolution 1, it means that the data can be perfectly modelled with a single linear model. If we only approach a score of 1 when the resolution gets close to n, it means that we need about n independent models to fit the data perfectly, hence about one model per point. This indicates the presence of non-linearities on small scales, i.e., noise typically. The maximal resolution for which we compute the deviation from linearity is $n/(d+2)$ where d is the dimension of the data after pre-encoding[2]. Obviously, a linear model in d dimensions can always fit perfectly in a neighborhood of size $k < d+2$ and

[2]To compute this linearity profile, we do not use zero padding nor separate representations for each feature. Instead, we simply concatenate all f_j's into one vector of size d.

we would thus always obtain a score of 1 at greater resolutions [3]. This is an important limitation of the proposed complexity measure: it is not applicable to datasets that have more dimensions than samples.

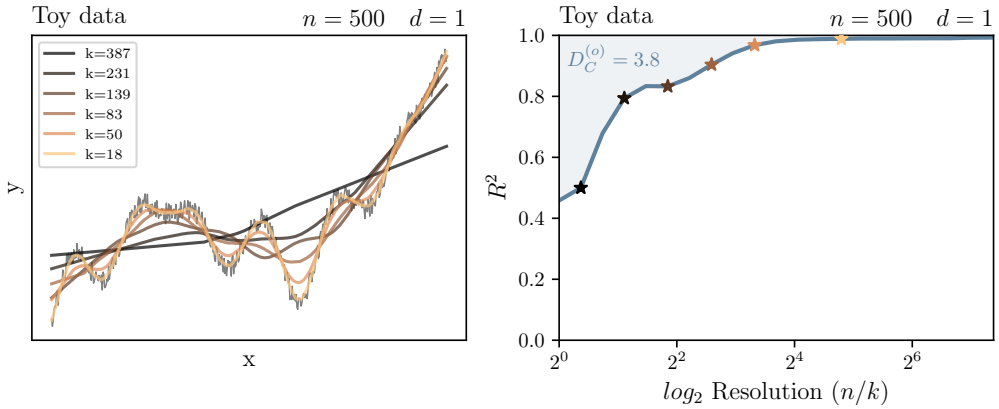


Figure 4: Linearity profile for a toy dataset. The plot on the left shows the various approximations of the original data obtained by fitting local linear models around each point in neighborhoods of decreasing size k. The plot on the right shows the curve obtained by computing R^2 between the various approximations and the original data. This amounts to computing the local deviation from linearity.

Finally, to summarize the deviation from linearity on different scales or resolutions, we propose to measure the observed data complexity $D_C^{(o)}$ as the area above the curve. If we compare again to perfectly linear data, the curve would culminate at 1 for every resolution and we would obtain $D_C^{(o)} = 0$. If we consider instead pure noise, the curve would drop close to 0 at every resolution and we would obtain $D_C^{(o)} \approx n/(d+2)$. This seems coherent with what we would expect from a data complexity measure: linear data would remain trivial to model no matter the sample size n whereas fitting pure noise would be increasingly difficult as n grows. We show the effect of adding noise to the toy data in Fig. 5; we can indeed observe that the curve falls down towards lower scores and that the complexity consistently increases.

Notice that our method for defining data complexity is rooted in the context of neural networks. Specifically, we focus on neural networks with

[3] More precisely, we would get a score of 1 for resolutions greater or equal than $n/(d+1)$, leaving a region between $n/(d+2)$ and $n/(d+1)$ which we cannot probe since our model considers only integer values for k by design.

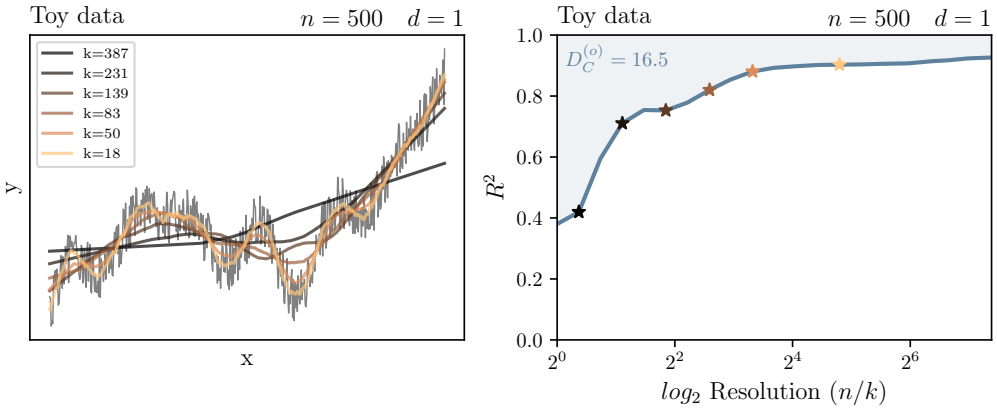


Figure 5: Linearity profile for the toy dataset with added noise. The plot on the left shows the various approximations of the original data obtained by fitting local linear models around each point in neighborhoods of decreasing size k. The plot on the right shows the curve obtained by computing R^2 between the various approximations and the original data. This amounts to computing the local deviation from linearity.

ReLU activations. Since ReLU is a piece-wise linear activation function, these networks model a mapping $\mathbb{X} \to \mathbb{Y}$ as a piece-wise linear function. This also implies that if a mapping is easily approximated with a piece-wise linear function, it will be easier to model for these neural networks. Whether or not a mapping can be well approximated by a piece-wise linear function is exactly what we try to measure, and since this is scale dependent, we measure it for a range of scales and take an average as our final data complexity measure.

4.2. Influence of d' and m' on effective model complexity

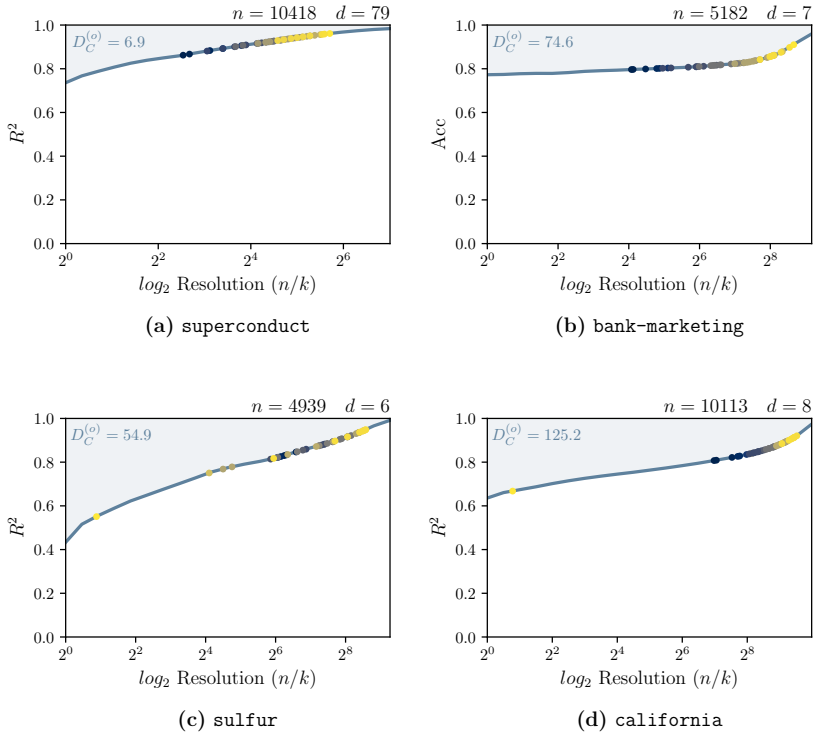
We can think of the expressive complexity of neural networks that have piecewise linear activation functions, like ReLU, in terms of the number of linear regions that they can produce to approximate a function [25]. The number of linear regions then reflects the capacity of the network. Building on our definition of data complexity as the non-linearity of the mapping between features and labels, we can also think about the effective complexity of a neural network as the non-linearity of the produced approximation function once it is fitted to a given dataset. In other words, the degree to which the linear regions are aligned or not with the neighboring ones. If all linear regions are perfectly aligned, the model approximates a linear function and has very low effective complexity. If all linear regions are oriented in different directions, the model has a very high effective complexity. According to

this view, the effective model complexity of our *feature-level* models will depend on two main factors: (1) the capacity through d' and m', and (2) a regularization element which will constrain the capacity, that is, force the alignment of linear regions.

To study this more formally, we assume that the non-linearity of the function learnt by training a neural network that has performance s^* on the training data is similar to the non-linearity corresponding to the point (r^*, s^*) on the linearity profile computed on the same training data (where r^* is the resolution corresponding to the score s^*). This assumption relies on the result that neural networks are naturally biased towards over-smoothed solutions [4]. Therefore, we can reasonably expect that the function learnt by training a neural network will resemble some smoothed approximation of the data obtained for a given resolution while computing the linearity profile. This allows us to project the training performance of our *feature level* models onto the linearity profile curve, retrieve the corresponding resolution values by simple interpolation, and use those resolutions as a proxy for the model effective complexity. We illustrate this for the four prototypical datasets in Fig. 6.

Upon first inspection, we observe that for a given dataset, models with higher capacity (higher d' and m') correspond to higher resolutions and thus, presumably, higher effective complexity. This is consistent with intuition. Moreover, for every dataset, we can plot resolutions corresponding to each model with respect to the data observed complexity, as shown in Fig. 7. We observe a direct relationship between resolution and data observed complexity. This is also expected: for a fixed and sufficient capacity, a model fitted to simpler data should have a lower effective complexity than if fitted to more complex data. Data complexity can thus be seen as the primary cause that regulates or tones down the potential capacity to the effective model complexity after training. This regularization driven by data obviously differs from explicit model regularization with additional objectives that constrain model parameters.

Closer inspection of Fig. 6 shows that the gain in resolution gets smaller as the capacity increases. Additionally, for some of the highest capacity models on the `sulfur` and `california` datasets, we observe a drop in training score and resolution. This is evident in Fig. 8, where we plot resolution with respect to d' and m'. This suggests a regularization effect that depends on the capacity itself. A plausible interpretation is that the linear regions pro-



(a) `superconduct` **(b)** `bank-marketing`

(c) `sulfur` **(d)** `california`

Figure 6: Model training scores projected on linearity profiles. Each point overlapping the linearity profile curve corresponds to a model with a different value for d' and m' (45 models per dataset). Points are colored according to parameter count (on a logarithmic scale): blue indicates lower parameter count; yellow indicates higher parameter count. The ordinate of a point reflects its corresponding model performance on the training data. Interpolation provides the abscissa of each point.

duced by a neural network with ReLUs are not independent from one another [25] and increasing the capacity of the network actually also increases this interdependence, thereby causing an implicit self-regularization phenomenon which strengthens with capacity. Under certain conditions of model architectures and data complexity, this self-regularization effect can get so strong that it leads to a significant decrease in training performance and model effective complexity. From our experiments, we observe that this phenomenon happens only for regression dataset and is favored by a high observed data complexity, a high ratio n/d (number of samples/dimension of the data) and

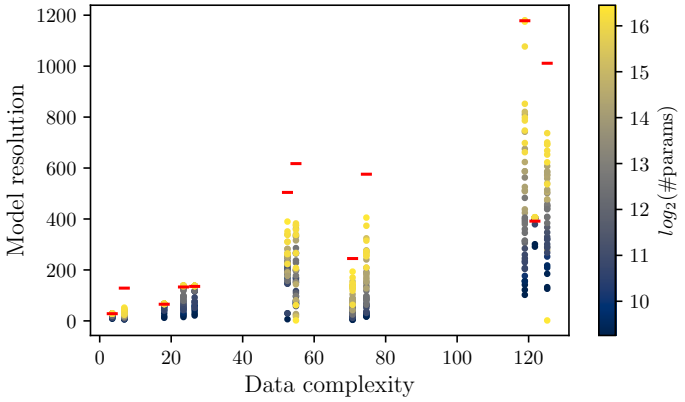


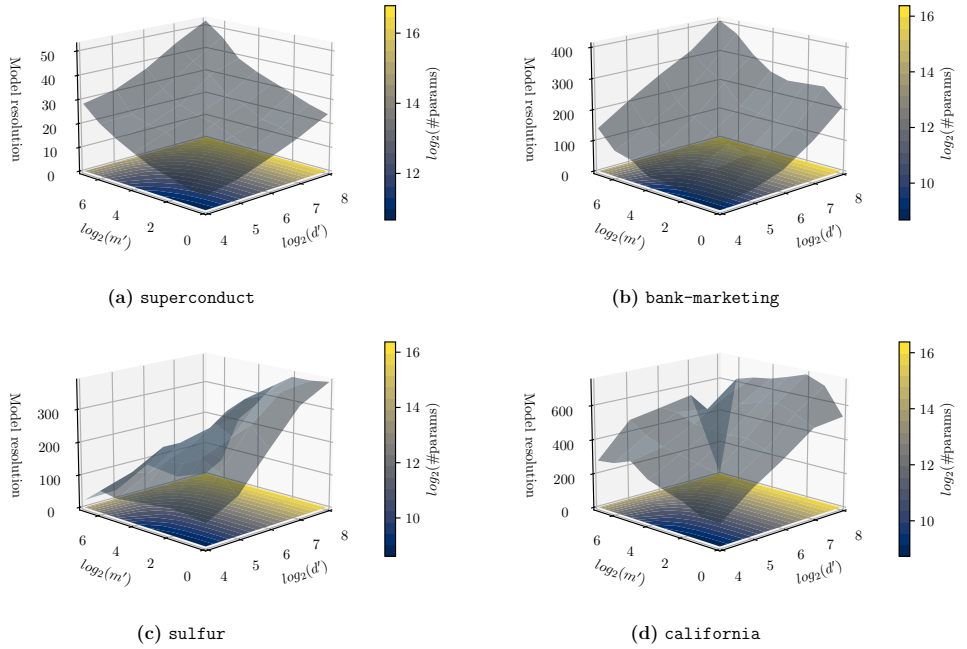
Figure 7: Model resolution versus data complexity. Each column of points corresponds to a dataset and each point within a column corresponds to a model with a different value for d' and m' (45 points per dataset). Points are colored according to parameter count (log scale). The red markers indicate for each dataset the maximal resolution $n/(d+2)$ for which our linearity profile is non-trivial.

a high value for m', the number of latent representations.

Although we do not prove its existence in any way, this self-regularization phenomenon is consistent with observations made in [15], reporting that neural networks for tabular data perform particularly worse than gradient boosted decision trees on very large datasets and/or dataset with a high ratio between number of samples and number of features. It also relates to the double descent phenomenon [26], where we would have models of higher capacity that are more regularized, thus effectively less complex, and leading potentially to better generalization performance.

4.3. Influence of d' and m' on generalization

As the issue of generalization revolves around both the model effective complexity and the data intrinsic complexity, studying generalization in this context is difficult because (i) we only have a proxy for effective model complexity and (ii) we do not have access to the intrinsic data complexity. In fact, the capability of the linearity profile to indicate the presence of small scale non-linearities is likeky to decrease when the data dimensionality of the data grows; in that case, the gap between intrinsic non-linearities and noise vanishes. It is hence particularly difficult to identify when we start fitting noise, and model effective complexity becomes higher than intrinsic



(a) `superconduct` (b) `bank-marketing`

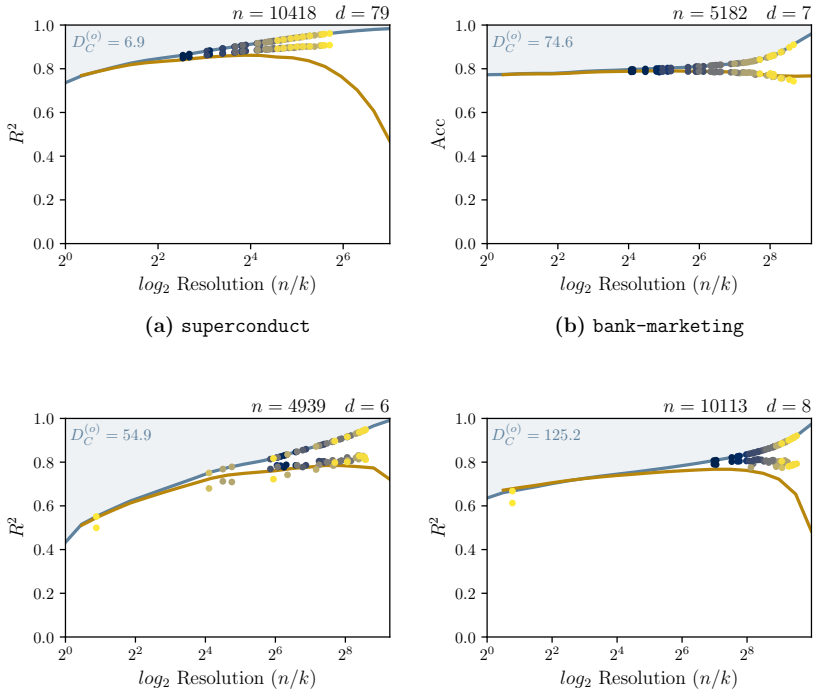
(c) `sulfur` (d) `california`

Figure 8: Model resolution versus m' and d'. The heat map indicates the magnitude of the number of trainable parameters.

data complexity.

However, we can still try to approximate the generalization error by computing a slightly modified version of the linearity profile: for a given point i and neighborhood of size k, instead of fitting the local linear model using both the k nearest neighbors and the point i itself, we fit the linear model using only the k nearest neighbors. The point i then serves as a size one local test set. When k is big, we do not expect to observe big differences between predictions made with or without point i at the local fitting stage. However, when k is small, differences can become important, especially if the data exhibits non-linearities and/or noise on small scales. A point $(\tilde{r}, \tilde{s})$ on the resulting curve can be interpreted as the expected test performance $\tilde{s}$ when we fit the data with a resolution equivalent to $\tilde{r}$. Figure 9 shows these approximated test performances as well as the observed test performance.

The approximated test performances reveals the over-fitting behavior, although it tends to overestimate the generalization error. Oddly, they are



(a) `superconduct` **(b)** `bank-marketing`

(c) `sulfur` **(d)** `california`

Figure 9: Linearity profiles and approximate test performance. The blue solid curve is the linearity profile. The gold solid curve is the modified linearity profile. Points that coincide exactly with the linearity profile curve correspond to the training performances of models, while points located beneath this curve represent the corresponding observed test performances. Both series of points are colored according to parameter count (log-scale): blue indicates lower parameter count; yellow indicates higher parameter count. The ordinate of a point is given by its corresponding model's performance on the training/test data. The abscissa of each point is found by interpolation of the linearity profile.

much more precise for classification datasets (only shown for `bank marketing` here). We further analyze the relationship between over-fitting behavior and model effective complexity by representing resolution of the model with respect to data complexity and generalization error in Fig. 10. We observe that models often over-fit more when the resolution is close to to the maximal resolution $n/(d+2)$ and that this happens more frequently with less complex data. This concurs with intuition: models with an effective complexity close or equal to the data observed complexity are more susceptible

to over-fitting, since the intrinsic data complexity is likely to be much lower than the observed complexity, especially in tabular data that are often irregular and noisy. Moreover, it is consistent with the reported observation that neural networks for tabular data are particularly sensitive to uninformative features [4]. Uninformative features would indeed artificially decrease data complexity and favor over-fitting. The fact that, in our experiments, over-fitting occurs less often for complex data sets can finally be put in relation to our previous observations about model effective complexity and our hypothesis of self-regularization. Indeed, self-regularization appears to be favored by high data complexity and would thus help mitigate over-fitting in this case.

Notice that this has implications for data pre-processing. It suggests that artificially increasing the complexity of the data would act as an implicit regularization and can be beneficial to avoid over-fitting. This is exactly the purpose of noise injection [27]. Complexity/noise should be added in a controlled way as not to widen the gap between intrinsic data complexity and observed data complexity, and potentially exacerbate over-fitting. Conversely, there is also an argument for exploring feature engineering techniques that lower data complexity. This leads to models with lower effective complexity and if the gap between observed complexity and intrinsic complexity is effectively narrowed, these feature engineering techniques also have the potential to mitigate over-fitting. However, as we do not have access to intrinsic data complexity, these considerations are not straightforward to apply and deserve further investigation that extends out of the scope of this study.

5. Avenues for future Investigation

While our approach to understanding data complexity and its relation to model effective complexity offers some valuable insights about model behavior and generalization, it also faces a few limitations. The most notable two limitations, in our view, are outlined below:

Limitation 1. By projecting the performances of our *feature-level* models onto the linearity profiles and subsequently using the corresponding resolutions as a measure of effective model complexity, we implicitly assume that two models performing equally on the training data would have the same effective complexity and also perform equally well (or badly) on the test set. However, this is neither obvious nor entirely supported by our experimental results. Indeed, when looking at the `sulfur` case, we observe models

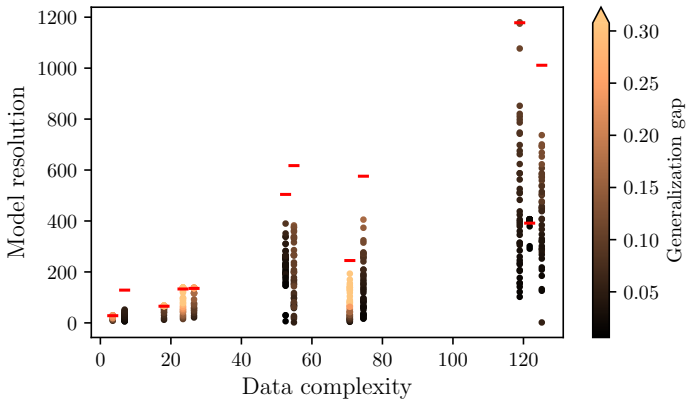


Figure 10: Model resolution versus data complexity and generalization error. Each column of points corresponds to a dataset and each point within a column corresponds to a model with a different value for d' and m' (45 points per dataset). Points are colored according to generalization error. The red markers indicate for each dataset the maximal resolution $n/(d+2)$ for which our linearity profile is non-trivial.

with equivalent training performances but significantly different test performances. It is also not necessary: in fact, a sensible measure of effective complexity would require that if two models perform the same on the training set, the one with the lower effective complexity would generalize better and perform best on the test set.

Limitation 2. It is also not obvious that we can project model performance on the linearity profile curves in the first place. While neural networks can indeed be reasonably assumed to behave at least similarly to a regularized collection of n local linear models, stating that they behave exactly like so might sound less reasonable. And it seems they do not indeed, as we observe differences between estimated over-fitting and empirically observed over-fitting, for instance.

These limitations prompt us to propose a more robust conceptual framework for exploring interplay between model complexity and data complexity. This framework is illustrated in Fig. 11. The idea would be to fit an arbitrarily large collection of independent local linear models to a data set and to apply an arbitrarily strong regularization that forces neighboring local linear models to be similar to one another. This is essentially the same idea as the linearity profile but whereas the latter has a fixed capacity of n models

that are increasingly regularized as the neighborhood size k becomes large; in this setting, capacity and regularization are independent. Each point in the capacity-regularization plane would correspond to a particular regularized collection of linear models and the curves that we have for the linearity profiles would become surfaces. Finally, instead of taking resolution as a proxy for effective complexity, we could take a function of regularization and capacity.

In light of this, the linearity profile is just a slice of fixed capacity of this more general view (curve in blue), whereas neural networks of increasing capacity would follow a curved path (in red) in the capacity-regularization plane. The higher the capacity and the higher m', the more curved the path; even so curved as to go up and down on the training surface, leading to models with high capacity but low effective complexity, as observed in our experiments. Moreover, within this framework, two points with the same training score could have different effective complexities and different test scores, thereby resolving the first limitation. The difference between the red curve and the blue curve resolves the second limitation and explains the difference that is observed empirically: a model on the blue path with a high training score would tend to have no regularization hence a potentially high over-fitting, while a neural network with high training score typically has high capacity and tends to be more regularized, hence less exposed to over-fitting.

6. Conclusion

This work has investigated the impact of dimension and number of latent representations in *feature-level* deep neural networks for tabular data. We demonstrate that varying these hyper-parameters yields distinct model behaviors during both training and testing phases, with these behaviors being intricately tied to data complexity. We have implemented a method to characterize data complexity and then used it to gain insights into the observed behaviors. We associate this with the concept of model effective complexity, and hypothesize a form of implicit self-regularization that intensifies with model capacity and the ratio of samples to dimensions. While this self-regularization phenomenon can alleviate over-fitting, it may also lead to diminished performance on training data. We believe that comprehending the intricate relationships between model and data complexity is pivotal for

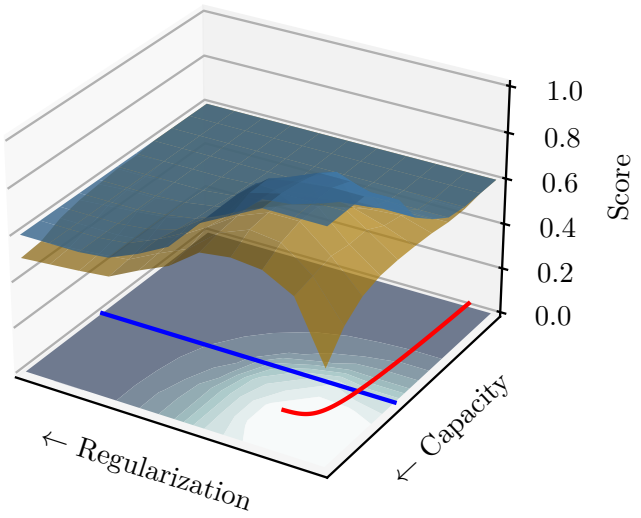


Figure 11: Proposed conceptual framework.

designing appropriate neural networks for tabular data, ultimately helping to understand the success or failure cases of deep learning.

Aknowledgments

PL is a FRIA grantee of the Fonds de la Recherche Scientifique-FNRS. JAL is a Senior Reasearch Associate with the Fonds de la Recherche Scientifique-FNRS. CdB is a beneficiary of a UCLouvain FSR Incoming Post-doctoral Fellowship.

Appendix A. Detailed network architecture

This detailed description of our network architecture is a complement to section 2 and 3. In our experiments, we use a three layer neural network. The first two layers form the encoder while the third layer is the decoder. The model is implemented in `pytorch 2.1.2` with `cu188` support.

The first layer takes as input a batch of pre-encoded samples, i.e., a tensor $\mathbf{e}$ of size (B, d, m) where B is the batch size, d the dimension of each encoded

feature, and m is the number of features. It outputs a tensor $\mathbf{h_0}$ of size (B, d', m') according to the equation :

$$\mathbf{h_0} = \text{ReLU}(W_0\tilde{\mathbf{e}} \ + \mathbf{b}_0\mathbf{1}^T) \ \text{ with } \ \tilde{\mathbf{e}} \ = \mathbf{e}A_0 \tag{A.1}$$

where $ReLU(\cdot)$ is the standard Rectified Linear Unit activation function, $\mathbf{W_0}$ is a trainable weight matrix of size $d' \times d$, $\mathbf{b_0}$ is a trainable bias vector of size $d' \times 1$, $\mathbf{1^T}$ is a vector of ones of size $1 \times m'$ and $\mathbf{A_0}$ is a trainable weight matrix of size $m \times m'$.

The second layer takes as input $\mathbf{h_0}$ and outputs $\mathbf{h_1}$, a tensor of size (B, d', m') according to the equation :

$$\mathbf{h_1} = \text{ReLU}(W_1\tilde{\mathbf{h}}_0 + \mathbf{b}_1\mathbf{1}^T) \ \text{ with } \ \tilde{\mathbf{h}}_0 = \mathbf{h}_0A_1 \tag{A.2}$$

where $\mathbf{W_1}$ is a trainable weight matrix of size $d' \times d'$, $\mathbf{b_1}$ is a trainable bias vector of size $d' \times 1$, $\mathbf{1^T}$ is a vector of ones of size $1 \times m'$ and $\mathbf{A_1}$ is a trainable weight matrix of size $m' \times m'$. The output of the second layer is then pooled using a standard sum operator to obtain a tensor $\mathbf{z}$ of size (B, d'). The tensor $\mathbf{z}$ is then passed to the last layer which outputs label predictions $\hat{\mathbf{y}}$ according to the equation:

$$\hat{\mathbf{y}} = \sigma(W\mathbf{z} + b) \tag{A.3}$$

where $\sigma(\cdot)$ is the identity in case of a regression task and a sigmoid in case of a classification task, $\mathbf{W}$ is a trainable weight matrix of size $1 \times d$ and b is a trainable scalar bias.

We then compute Mean Square Error in case of a regression task, and Binary Cross Entropy in case of a classification task. The loss is averaged over the batch.

We use the pytorch Adam optimizer with a learning rate of 0.001. Other parameters of the optimizer are left to default values. The batch size B is fixed to 256 and the number of epochs is 200 for each experience.

The value of d and m are dependent on the data set. The value of d' is taken from the set $\{16, 32, 64, 128, 256\}$ while the value for m' is taken from the set $\{1, 2, 4, 8, 16, 32, 64, 128\} \cup \{m\}$.

References

[1] T. Chen, C. Guestrin, XGBoost: A Scalable Tree Boosting System, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, San Francisco California USA, 2016, pp. 785–794. doi:10.1145/2939672.2939785.

[2] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, A. Gulin, CatBoost: unbiased boosting with categorical features, in: Advances in Neural Information Processing Systems, Vol. 31, Curran Associates, Inc., 2018.

[3] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, LightGBM: A Highly Efficient Gradient Boosting Decision Tree, in: Advances in Neural Information Processing Systems, Vol. 30, Curran Associates, Inc., 2017.

[4] L. Grinsztajn, E. Oyallon, G. Varoquaux, Why do tree-based models still outperform deep learning on typical tabular data?

[5] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, G. Kasneci, Deep Neural Networks and Tabular Data: A Survey, arXiv:2110.01889 [cs] (Jun. 2022).

[6] Z. Li, Z. Cui, S. Wu, X. Zhang, L. Wang, Fi-GNN: Modeling Feature Interactions via Graph Neural Networks for CTR Prediction, in: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, 2019, pp. 539–548, arXiv:1910.05552 [cs]. doi:10.1145/3357384.3357951.

[7] K. Zhou, Z. Liu, R. Chen, L. Li, S.-H. Choi, X. Hu, Table2Graph: Transforming Tabular Data to Unified Weighted Graph, in: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, Vienna, Austria, 2022, pp. 2420–2426. doi:10.24963/ijcai.2022/336.

[8] W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, M. Zhang, J. Tang, AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks, in: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM '19, Association

for Computing Machinery, New York, NY, USA, 2019, pp. 1161–1170. doi:10.1145/3357384.3357925.

[9] S. O. Arik, T. Pfister, TabNet: Attentive Interpretable Tabular Learning, arXiv:1908.07442 [cs, stat] (Dec. 2020).

[10] X. Huang, A. Khetan, M. Cvitkovic, Z. S. Karnin, TabTransformer: Tabular Data Modeling Using Contextual Embeddings, ArXiv (Dec. 2020).

[11] Y. Gorishniy, I. Rubachev, V. Khrulkov, A. Babenko, Revisiting Deep Learning Models for Tabular Data, arXiv:2106.11959 [cs] (Nov. 2021).

[12] G. Somepalli, M. Goldblum, A. Schwarzschild, C. B. Bruss, T. Goldstein, SAINT: Improved Neural Networks for Tabular Data via Row Attention and Contrastive Pre-Training, arXiv:2106.01342 [cs, stat] (Jun. 2021). doi:10.48550/arXiv.2106.01342.

[13] J. Yan, J. Chen, Y. Wu, D. Z. Chen, J. Wu, T2G-FORMER: Organizing Tabular Features into Relation Graphs Promotes Heterogeneous Feature Interaction, Proceedings of the AAAI Conference on Artificial Intelligence 37 (9) (2023) 10720–10728, number: 9. doi:10.1609/aaai.v37i9.26272.

[14] A. Kadra, M. Lindauer, F. Hutter, J. Grabocka, Well-tuned Simple Nets Excel on Tabular Datasets, in: Advances in Neural Information Processing Systems, Vol. 34, Curran Associates, Inc., 2021, pp. 23928–23941.

[15] D. McElfresh, S. Khandagale, J. Valverde, V. P. C, B. Feuer, C. Hegde, G. Ramakrishnan, M. Goldblum, C. White, When Do Neural Nets Outperform Boosted Trees on Tabular Data?, arXiv:2305.02997 [cs, stat] (Oct. 2023).

[16] E. Couplet, P. Lambert, M. Verleysen, J. A. Lee, C. De Bodt, On the number of latent representations in deep neural networks for tabular data, ESANN proceedings 1 (2023) 1.

[17] T. K. Ho, M. Basu, Complexity measures of supervised classification problems, IEEE transactions on pattern analysis and machine intelligence 24 (3) (2002) 289–300.

[18] A. I. Maciel, I. G. Costa, A. C. Lorena, Measuring the complexity of regression problems, in: 2016 International Joint Conference on Neural Networks (IJCNN), IEEE, 2016, pp. 1450–1457.

[19] J. Zubek, D. M. Plewczynski, Complexity curve: a graphical measure of data complexity and classifier performance, PeerJ Computer Science 2 (2016) e76.

[20] H. Liu, K. Simonyan, Y. Yang, Darts: Differentiable architecture search, arXiv preprint arXiv:1806.09055 (2018).

[21] L. Hu, Z. Wang, H. Li, P. Wu, J. Mao, N. Zeng, -darts: Light-weight differentiable architecture search with robustness enhancement strategy, Knowledge-Based Systems (2024) 111466.

[22] Y. Gorishniy, I. Rubachev, A. Babenko, On embeddings for numerical features in tabular deep learning, Advances in Neural Information Processing Systems 35 (2022) 24991–25004.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[24] X. Hu, L. Chu, J. Pei, W. Liu, J. Bian, Model complexity of deep learning: a survey, Knowledge and Information Systems 63 (10) (2021) 2585–2619. doi:10.1007/s10115-021-01605-0.

[25] G. F. Montufar, R. Pascanu, K. Cho, Y. Bengio, On the Number of Linear Regions of Deep Neural Networks, in: Advances in Neural Information Processing Systems, Vol. 27, Curran Associates, Inc., 2014.

[26] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, I. Sutskever, Deep double descent: where bigger models and more data hurt*, Journal of Statistical Mechanics: Theory and Experiment 2021 (12) (2021) 124003, publisher: IOP Publishing and SISSA. doi:10.1088/1742-5468/ac3a74.

[27] O. Dhifallah, Y. Lu, On the inherent regularization effects of noise injection during training, in: International Conference on Machine Learning, PMLR, 2021, pp. 2665–2675.