Do We Always Need the Simplicity Bias? Looking for Optimal Inductive Biases in the Wild

Supplementary Material

A. Reviewers' FAQ

This section contains interesting questions raised during the review of this paper (paraphrased) and our answers.

Why use MLPs instead of CNNs or ViTs for example? The choice of **unstructured MLPs** is deliberate. Since the primary goal is to discover optimal inductive biases via optimization, it makes sense to start with architectures that impose little initial constraints.

Can the proposed method for learning activation functions be applied to other architectures? In principle yes, but the bi-level optimization is expensive. We did not attempt to use it with large models. This method is meant as an exploratory tool, and the insights it delivered are much more fundamental. They could serve in the design/selection of future architectures independently of this optimization method. For example, Teney et al. [96, Fig. 5] already evaluated how various components (e.g. attention) can nudge inductive biases in ways similar to activation functions.

Why is the scale of TV values different across datasets? TV values are not comparable across datasets because of the different distances between data points in input space.

Are learned activation functions more akin to pre-trained initialization than architecture choices? Not really, because initializations can vanish with enough iterations of fine-tuning, while the effects of activation functions remain. However, it is true that parametrized activations carry more information than typical architecture choices.

B. Related Work

Inductive biases in deep learning are due to choices of architecture [33] and of the learning algorithm (optimizer, objective, regularizers [50]). We focus on the former. The simplicity bias has been studied from both aspects. Most explanations attribute it to loss functions [70] and gradient descent [7, 40, 60, 92]. But work on untrained networks shows that it can be explained with architectures alone [15, 31, 64, 96, 99]. Teney et al. [96] showed that the choice of activation function can modulate the simplicity bias. The **spectral bias** [48, 75] or frequency principle [104] is a related but different effect related to training dynamics: NNs approximate low-frequency components of the target function earlier during training with SGD.

Suitability of the simplicity bias. The tendency of NNs to represent simple functions is thought as the key why over-parametrized networks avoid overfitting [8, 71]. Schmidhuber [83] even proposed to regularize a model's Kolmogorov complexity to improve generalization. The preference for simplicity aligns with **Occam's razor**, a philosophical principle whose (absence of) justification has long been debated [66, Appendix A]. Domingos [19] discussed arguments against Occam's razor for knowledge discovery.

Side-effects of the simplicity bias. The simplicity bias is responsible for **shortcut learning** [29, 74, 93] and for amplifying performance disparities [9]. A vast literature addresses shortcut learning with alternative losses [74], architectures [41], diversification mechanisms [1, 94, 95], etc. No study has however addressed its root cause, which we pinpoint to architectural choices, activation functions in particular. The simplicity bias is also detrimental in the use of NNs for **scientific computing** such as solving PDEs [106, Section 5.4]. A solution relevant to activation functions was proposed in MscaleDNNs [57] by restricting them to a compact support. The simplicity bias makes it difficult for **implicit neural representations** to represent sharp image edges for example [78]. The prevailing solution is to replace activation functions with sines [88], Gaussians [77], or wavelets [80]. Fourier features [86] are another solution, in fact mathematically equivalent to periodic activations [103, Sect. 5]. With **tabular data**, NNs are known to often perform poorly [20, 35]. Solutions include Fourier features and numerical embeddings [32, 55] which can be seen as special cases of learned activation functions. In **reinforcement learning**, a few studies have suggested that the spectral bias of typical architectures may be suboptimal [53] and have experimented with Fourier features [107] and sine activations [62]. These examples support our message that the simplicity bias is not always desirable. They also support the search for new activation functions to modulate it.

Activation functions are key for introducing non-linearities in NNs. Many options were considered early on, e.g. sine activations in the Fourier Neural Networks from 1988 [27]. ReLUs are often credited for enabling the rise of deep learning by avoiding vanishing gradients [61]. However they are also essential in inducing the simplicity bias [96] which may be just as important. The research community has slowly converged towards smooth handcrafted variants of ReLUs such as GeLUs [21, 39, 76]. Some works proposed to **learn activation functions** using extra parameters optimized alongside the weights of the network [2, 5, 6, 11, 13, 22, 45, 82, 91]. See Jagtap and Karniadakis [44] for a comprehensive review. The goal is to better fit the training data with an activation function that can

evolve during training. In contrast, we use meta learning to find an activation function that induces better inductive biases, such that training with this *fixed* activation provides better generalization. This requires bi-level optimization, episodic training, and unbiased parametrization that allows us to learn activations very different from existing ones. **Kolmogorov-Arnold Networks** [59] parametrize the connections in a NN, which is equivalent to learning different activation functions across channels and layers. They use a parametrization as splines similar to ours. Their benefits in physics-related problems likely result from the alterations to the inductive biases studied in this paper. Our method differs from **neural architecture search** [100] in its ability to discover novel activation functions from scratch, rather than selecting from predefined candidates [91] or restricted parametric functions [3].

C. Method for Learning Activation Functions

This section provides details about the proposed method.

Novelty. Our method is designed to support an analysis of inductive biases and their effects in two steps.

1. **Learning an activation function** optimized for generalization on a specific dataset.
2. **Using this new fixed activation function** to train a network "as usual", such that the trained model can be analyzed and compared with any other e.g. a baseline ReLU architecture.

Our method is therefore very different from most existing works about learning activation functions [2, 5, 6, 11, 13, 22, 45, 82, 91]. These usually train the model weights and activation function together for the same objective i.e. fitting the training data. In our formulation, the activation function is trained for a different objective i.e. maximizing *generalization*. We exploit this in Section 3.4 (Shortcut Learning) by simulating in-domain (ID) and out-of-distribution (OOD) conditions. Each setting then learns a different activation function that prioritizes the learning of different features.

Parametrization as splines. We parametrize the learned activation functions as splines such that we can learn function with arbitrary, irregular shapes if needed. This contrasts with existing works on the learning of activation functions that constrain the search e.g. to combinations of existing activations [91], a small MLP [5], or other parametric functions [3]. A parametrization as splines was already used by Scardapane et al. [81] and in work concurrent to ours on Kolmogorov-Arnold networks [59]. Some technical details:

- The parametrization takes three hyperparameters n_c, a, b.
- n_c specifies the number of control points, typically 50.
- The control points are spread regularly in the $[a, b]$, typically $[-5, +5]$ to cover typical activation values.
- A spline then represents piecewise linear segments that interpolate the values specified in the parameters $\boldsymbol{\psi} := [\, g_{\boldsymbol{\psi}}(a), \dots g_{\boldsymbol{\psi}}(b)) \,] \in \mathbb{R}^{n_\text{c}}$. Outside $[a, b]$, g extrapolates the values of $g(a)$ and $g(b)$.
- In our exploratory work, we compared this piecewise linear version with cubic splines, which are smoother but computationally more expensive. Both performed similarly. We also compared it with a faster nearest-neighbor interpolation of control points. This performed much worse than the piecewise linear version.

Implementation of the algorithm. We reproduce the complete procedure below. The model $f_{\boldsymbol{\theta},\boldsymbol{\psi}}$ represents any chosen architecture with weights/biases $\boldsymbol{\theta}$ and activation functions parametrized by $\boldsymbol{\psi}$. The gradient updates $\text{GD}(\cdot,\cdot)$ are described as full-batch updates, but they can be implemented with any optimizer e.g. mini-batch SGD or Adam.

Algorithm 1 Meta-learning an activation function (AF).

Input: training data $\mathcal{T}$; untrained neural model $f_{\boldsymbol{\theta},\boldsymbol{\psi}}$
Initialize $\boldsymbol{\psi}$ with zeros *Parametrization of AF*
$n_\text{tr} \leftarrow 0$ *Number of inner-loop iterations*
while $n_\text{tr} < n_\text{tr}^\text{max}$ ***Outer loop: train AF***
  Increment n_tr
  Sample the episode's tr. ($\mathcal{T}'$) and val. ($\mathcal{V}$) sets from $\mathcal{T}$
  Initialize $\boldsymbol{\theta}$ randomly *Model weights and biases*
  for n_tr **steps** ***Inner loop: train model with fixed AF***
    Eval. loss on $\mathcal{T}'$: $L \leftarrow \Sigma_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{T}'} \mathcal{L}\big(f_{\boldsymbol{\theta},\boldsymbol{\psi}}(\boldsymbol{x}, \boldsymbol{y})\big)$
    Gradient step on weights/biases: $\boldsymbol{\theta} \leftarrow \text{GD}(\boldsymbol{\theta}, \nabla_{\boldsymbol{\theta}} L)$
  Eval. loss on $\mathcal{V}$: $L \leftarrow \Sigma_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{V}} \mathcal{L}\big(f_{\boldsymbol{\theta},\boldsymbol{\psi}}(\boldsymbol{x}, \boldsymbol{y})\big)$
  Gradient step on AF: $\boldsymbol{\psi} \leftarrow \text{GD}(\boldsymbol{\psi}, \nabla_{\boldsymbol{\psi}} L)$
  if performance on $\mathcal{V}$ worsens **then** break *Early stopping*
Output: optimized AF $\boldsymbol{\psi}$

The bi-level optimization is expensive since every outer iteration trains the model from scratch. We mitigate this as follows. First, we train small-width models. Section 3.3 shows that the learned activations subsequently transfer to wider models. Second, we do not train the model to convergence in the inner loop. Instead, we progressively increase the number of inner iterations. This reduces the computational expense and makes the inner task progressively harder. Third, second-order derivatives (i.e. backpropagating through the inner gradient updates) are only computed over the last t inner steps (typically t=5). Our exploratory work found this to be better than a complete linearization (no second-order derivatives) and vastly cheaper than backpropagating through the whole inner loop (which was not even testable at all because of the required GPU memory).

Optimization. The optimization of the activation function in Algorithm 1 proved to be a very difficult non-convex

problem with many local minima. We tried various optimizers for the gradient updates on ψ (SGD with and without momentum, RMSprop). No option was consistently better. We also tried to run multiple instances of the inner loop in parallel (with several models initialized differently) to stabilize the gradients $\nabla_\psi L$. However this usually provides worse solutions, indicating that exploration is indeed beneficial to avoid local minima.

A simple but effective workaround is to use vanilla gradient descent with restarts, i.e. running Algorithm 1 with a different:

- random seed,
- learning rate to update ψ in $[0.01, 0.2]$,
- number of control points $n_c \in [50, 400]$,
- number of inner steps backpropagated through $t \in [1, 50]$,
- initialization as zeros or as a ReLU.

This is enough to learn slightly different activation functions. We then keep the best one according to its performance on the validation set after using it to retrain a model from scratch (as a fixed activation function) .

D. Ablations of the Proposed Method

We evaluate below the design choices of the proposed method to learn activation functions. We perform these experiments on the image regression task with FASHION-MNIST and 1-hidden layer MLPs We report averages and standard deviations over 10 random seeds. See Section E.2 for other experimental details. See the captions of Tables 1–3 for the takeaways of each experiment.

Table 1. Evaluation of the variance across runs (over 10 random seeds and 4 restarts). It is quite similar for the baseline and the learned-activation models. The latter models obtain a higher accuracy on average. These results verify that the improvements from the learned activations are not simply due to running more trials with more chances of finding a "lucky run". We also show that the restarts (i.e. running the optimization with multiple hyperparameters, see Section C) help find a better solution but are not indispensable to obtain an improvement over the baseline.

Activation function	Accuracy (%)
ReLU baseline	53.1 $\pm$ 0.4
Learned, average across restarts / hyperparameters	56.6 $\pm$ 0.7
Learned, **best** across restarts / hyperparameters	**57.2** $\pm$ 0.5

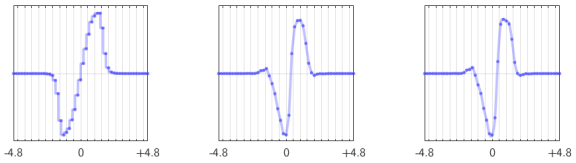

Figure 15. Activation functions learned with different interpolation methods. The linear and cubic ones are nearly identical.

Table 2. Evaluation of different interpolation methods to represent learned activation functions. The nearest-neighbor interpolation is cheap to evaluate but performs the worst. The linear one (used in all our experiments) is almost identical to the cubic one (in appearance and performance) while being faster to evaluate.

Activation function	Accuracy (%)
ReLU	53.1 $\pm$ 0.4
Learned, nearest-neighbor interpolation	55.1 $\pm$ 0.9
Learned, linear spline (default)	57.2 $\pm$ 0.5
Learned, **cubic** spline	**57.3** $\pm$ **0.7**

Table 3. Evaluation of different outer-loop objectives. The naive version simply optimizes the activation for minimum loss on the *training* data, but this is suboptimal. Ideally, one would like to optimize the loss on the *test* data (which would require cheating by accessing the test labels). We approximate it by optimizing on held-out *validation* data, which the results show to be about as good (the last row would is expected to be the best without any evaluation noise).

Activation function	Accuracy (%)
ReLU	53.1 $\pm$ 0.4
Learned to minimize loss on training data (naive)	56.7 $\pm$ 0.3
Learned to minimize loss on **validation** data (default)	**57.2** $\pm$ 0.5
Learned to minimize loss on test data (cheating)	56.9 $\pm$ 0.5

E. Experimental Details & Additional Results

E.1. General Experimental Details

When training MLPs on a given dataset, we first **tune standard hyperparameters** for the best validation accuracy using ReLUs (optimizer, batch size, learning rate). We reuse these hyperparameters for all other experiments on this dataset, i.e. we do not tune them again for the learned activation functions. Every experiment uses **early stopping** i.e. we keep the model at the training step with the best validation accuracy.

All experiments were run on a single laptop (Dell XPS 15) with an Nvidia RTX 3050 Ti (4 GB of GPU memory).

Variance in the results. In order to make the analysis of results stable and consistently reproducible, we use two interventions that greatly reduce the variance across seeds and training iterations. First, the models are trained with large- or full-batch gradient descent (typically 4096 examples per mini-batch). This eliminates most of the variation across seeds. Second, we use a simple stochastic weight averaging (SWA). That is, when evaluating a model, we use the average of the optimized weights over the last 50 training steps. This consistently improves the accuracy of all models, but it does not alter the training trajectories (by design) and we verified that it does not alter the ranking of models. The main advantage here is that it greatly stabilizes the performance across training iterations, i.e. the training curves are much smoother hence easier to analyze.

E.2. Image Datasets

Data. We use slightly cropped versions of the images in the original datasets. This makes the data and models smaller and allows us to run a larger number of experiments with limited computational resources. This makes the tasks slightly more difficult, hence the accuracies being lower baselines reported in prior work. For MNIST, we crop 5 pixels on every side. For SVHN, we crop 8 pixels on each of the the left and right sides.

Architecture. We use fully-connected MLP. Given that our goal is to evaluate the inductive biases induced by the choice of activation function, MLPs minimizes the possible interactions with other architectural components that would complicate the analysis.

The only improvement over vanilla MLPs is the inclusion of **residual connections**. After each hidden layer, the output of the activation function is summed with the input to the layer (from before the application of weights and biases). This never hurts the accuracy, and helps when learning different layer-specific activations functions.

For each dataset, we trained MLPs with 1 to 4 hidden layers, both with ReLUs and learned activation functions. Our main results retain the MLP whose depth is best for each activation function. We provide in Figure 17 the full results for every depth. We can see that the best number of layers is sometimes different across activation functions.

Regression tasks. We use the same data as the image classification tasks. The ground-truth regression targets are the class IDs $\{0, 1, ..., 9\}$ that we normalized to $[-1, 1]$. I.e. we assign to the classes values regularly spread within $[-1, 1]$. This normalization is standard practice for regression models to make the optimization numerically easier. The MLP models output a single scalar with their last layer with no softmax or sigmoid.

Additional results. We provide below results on all four image datasets. The main paper only includes results on MNIST for space reasons, but similar observations can be made on the others.

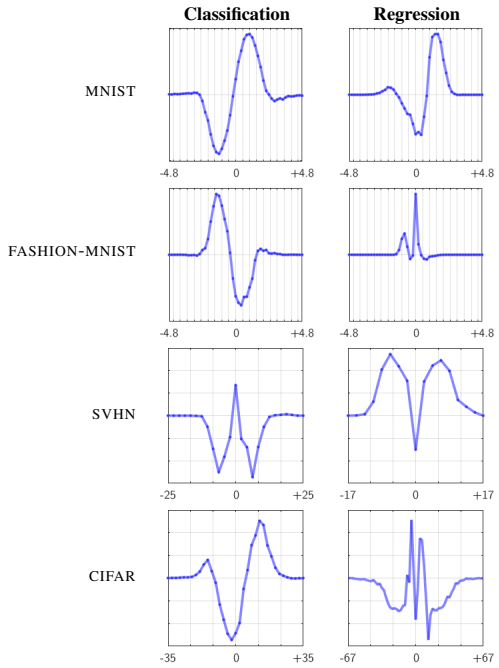

Figure 16. Activation functions learned for image datasets treated as classification or regression tasks. The activation functions learned for regression contain more irregularities. These help networks represent complex functions with sharp transitions.

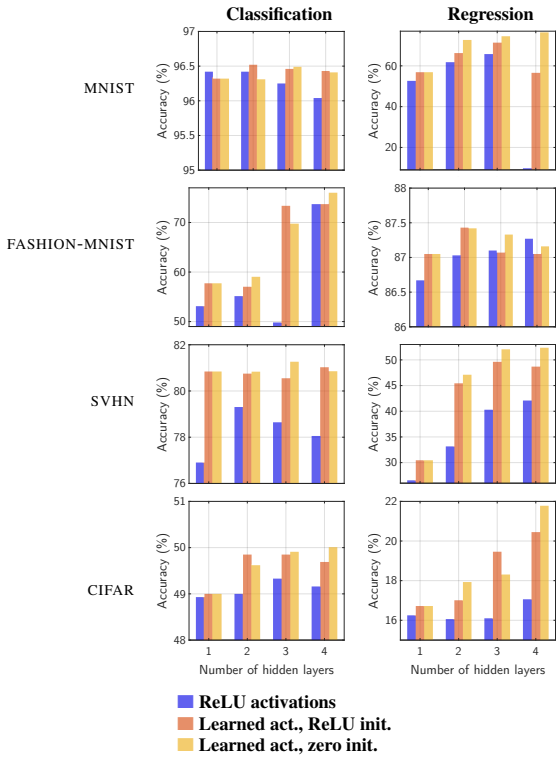

Figure 17. Image datasets, results per number of layers.

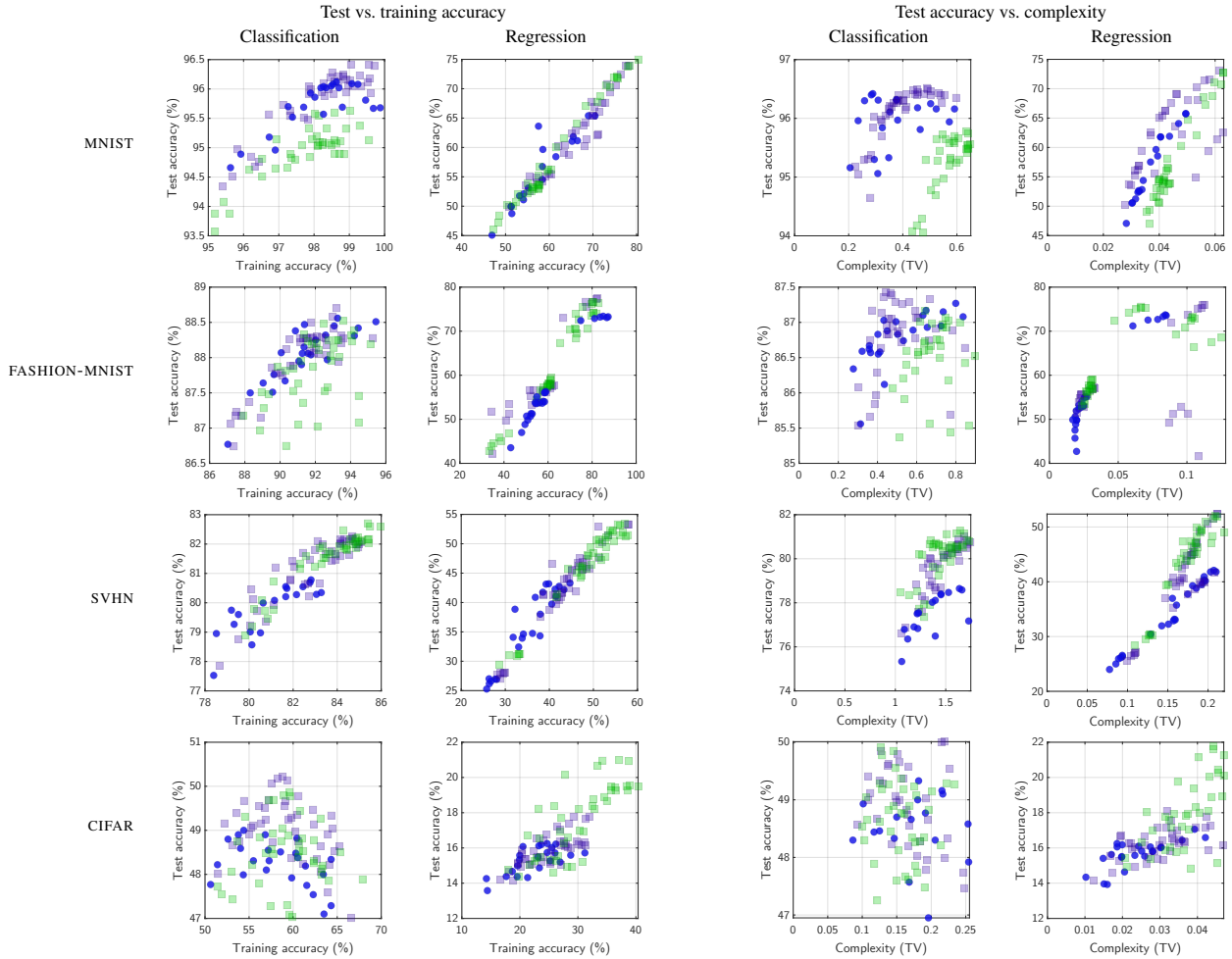

Figure 18. Analysis of models trained on image datasets. Each marker represents a model with different hyperparameters or number of training steps, and ReLUs (●) or learned activations with initialization as ReLUs (■) or as zeros (■). (Left) tr/test acc, models with learned activations have better accuracy than ReLUs, especially those learned from a random initialization.

E.3. Transfer of Learned Activation Functions across Image Regression Datasets

We provide below additional results on the transfer of learned activations across datasets, using the image regression tasks and 4-hidden layer MLPs. As in most experiments, we train a dataset-specific activation on each dataset (MNIST, FASHION-MNIST, CIFAR, SVHN) then use each of them to train a different model on each dataset. This gives a 4×4 matrix of results (middle rows in Table 4). We also attempt to learn an activation function on all datasets simultaneously (last row). See the table caption of the observations.

Table 4. Transfer of learned activation functions across image regression datasets. The diagonal elements (**gray cells**) correspond to activation functions optimized for a specific dataset then used to train a model on the same dataset. These obviously work well, but other combinations also surpass the ReLU baseline, which indicates positive transfer across datasets. The one learned on all datasets (**last row**) only improves over ReLUs on the two harder datasets (SVHN, CIFAR) and the improvements are (expectedly) smaller than with dataset-specific solutions. Further work may be needed to better balance multiple tasks when learning an activation function for multiple datasets.

Activation function	Accuracy (%) of models trained on MNIST	FASHION-M.	SVHN	CIFAR	Average Δaccuracy compared to ReLU
ReLU	76.7	73.9	42.9	16.1	0.0
Learned on MNIST	**79.7**	73.0	41.0	18.2	**+0.6** $\pm$ 2.3
Learned on FASHION-MNIST	64.3	**75.1**	39.7	**23.7**	-1.7 $\pm$ 8.4
Learned on SVHN	61.0	73.2	**54.1**	19.2	-0.5 $\pm$ 11.3
Learned on CIFAR	57.6	74.5	41.0	22.6	-3.5 $\pm$ 11.0
Learned on all datasets simultaneously	76.2	72.8	45.0	17.4	+0.4 $\pm$ 1.5

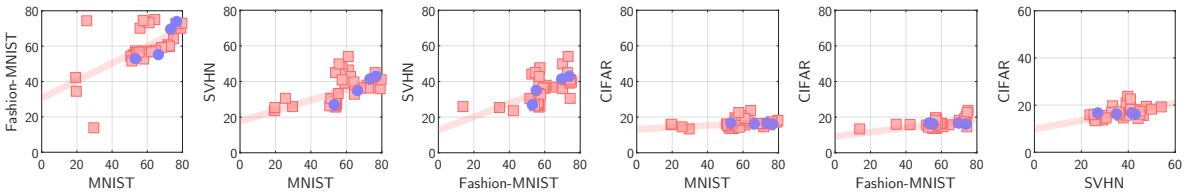

Figure 19. Transfer across image regression datasets. Each marker represents an MLP architecture with 1 to 4 hidden layers, with ReLUs (●) or activation functions learned on each of the four datasets (■). We plot the accuracy of each architecture on pairs of datasets to show that improvements often correlate across datasets (the line represents the best linear fit to the ■).

E.4. Tabular Data

Implementation details

- **Nearest-neighbor (k-NN)**. We use Matlab's implementation `fitcknn()` with Bayesian hyperparameter optimization for the number of neighbors and the distance measure (L1 or L2).
- **Boosted trees**. We use Matlab's implementation `fitcensemble()` with Bayesian hyperparameter optimization of standard hyperparameters. All the tabular datasets that we use are binary classification tasks, and the classification trees therefore produce discrete class predictions. To make the visualizations of "soft predictions" as in Figure 7c, we also train *regression* trees with `fitrensemble()`, using the class labels in $\{0, 1\}$ as regression targets. The output of these regression trees is then more comparable to the outputs (logits) from the MLP models.
- **Linear model**. We implement this baseline with the same code as our MLP models but with no hidden layer.
- **MLP models**. Our models use 1 to 4 hidden layers, a width of 256, and they are trained with RMSprop [97] with large mini-batches of 4096 examples to provide stable and consistent results. The number of layers is selected for the best validation accuracy for each type of activation function. The learning rate is also selected for best validation accuracy, but only once with ReLUs then reused for other activation functions.
  The performance of our models would likely improve with additional hyperparameter tuning. The **width** alone has a large impact on accuracy, as evaluated in Figures 8 and 21. Our goal is not best absolute performance so we did not expend resources in hyperparameter tuning and focused on **like-for-like comparisons** (i.e. only changing the activation function). If anything, our MLP models (and those with learned activation functions in particular) are at a disadvantage compared to the baselines.
- The **TanH activation functions with a prefactor** follow Teney et al. [96]. They are simple TanH functions with a multiplier: $\tanh(\alpha x)$. The scalar $\alpha \in [0.01, 8]$ is tuned for the best validation accuracy and shared across layers. The learning rate λ, which is originally tuned on the ReLU model as mentioned above, is adjusted as $\lambda \leftarrow \lambda/\alpha$.
- **Data normalization**. For every tabular dataset, we normalize the data (shift and scale) such that every input dimension ("column" in tabular terms) occupies the $[-1, 1]$ range. We experimented with other options: a normalization to unit standard deviation, and a quantile normalization to approximate a Gaussian or uniform distribution for every dimension. However they produced disparate results across our 16 datasets, so we settled with the simplest option to keep things consistent. If absolute performance is the objective, this should be optimized for each dataset. It has a large effect on the accuracy of MLPs, but not of trees nor k-NN classifiers. So again, our models are likely at a disadvantage compared to the baselines.

Details on the visualizations

- In Figure 7c, the grayscale images are produced by evaluating each model on $200 \times 200 = 40,000$ points in a 2D slice of the input space defined as follows. We first select one training example $\boldsymbol{x}$ at random. We then select two input dimensions m, n at random. We create every point of the slice by replacing the mth and nth values (the scalars $x[m]$ and $x[n]$). of $\boldsymbol{x}$ by every possible value in a grid of 200×200 values over $[-1, 1] \times [-1, 1]$. Since our data is normalized such that every dimension covers $[-1, 1]$, we now have a slice of inputs in a realistic range. This also explains why the training examples, marked by ·in Figure 7c are not centered in the images. They would be centered only if $x[m] = x[n] = 0$.
  The values plotted as a grayscale image are the network's output before a softmax/sigmoid activation. These values are not bounded to a specific range, so we scale them in each image to fill the black $\rightarrow$ white range.
- In Figure 7a and b, the loss and complexity landscapes are produced by evaluating models over a 50×50 grid covering a 2D slice of the parameter space (weights and biases). The slices are chosen to align with the first two principal directions of the trajectory. We obtain them by computing the PCA of a matrix made of the parameters from a number of checkpoints recorded over the training of the model. The 50×50 sets of parameters are obtained by applying perturbations to the trained model along these two directions. For each such set of parameters, we evaluate the model's training loss and its complexity (Section F) to make the loss and complexity landscapes. The range of loss/complexity values is consistent across all the visualization of a given dataset (i.e. a given color represents the same level of complexity across all plots in Figure 7b for example).

Intuition for the "input activation functions" (IAFs). The IAFs are activation functions that are applied directly on the input data, before it is passed to a standard MLP. The key is that these IAFs are applied independently to each dimension, such that they can implement a different behavior for each dimension (or "column" of the data). This is particularly useful for tabular data because every dimension can represent a different type of information. In comparison, once the data is passed

through the first layer of an MLP, the dimensions are all mixed together, and the subsequent activation function(s) are applied similarly to every dimension of the hidden representations.

The property of tabular datasets of requiring little or sparse feature interaction is well known and has been exploited in prior architectures designed for tabular data, see e.g. Gorishniy et al. [32]. This property is also a likely reason why decision trees are well suited to tabular data, since they implement decision boundaries aligned with dimensions of the data.

Additional results. See the figure below and their captions for details and observations. In Figures 22 and 23, each marker represents a model with different hyperparameters, number of training steps, and ReLUs (●) or learned activations initilized as ReLUs (■) or as zeros (■). The k-NN and tree models are represented as ♦ and ▲.

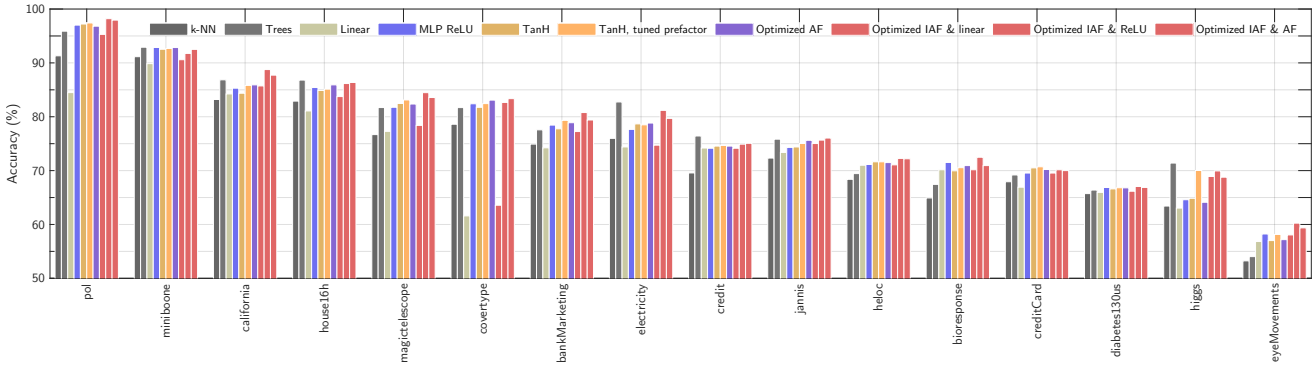

Figure 20. Comparison of model types on every tabular dataset [34], approximately sorted by decreasing performance. In almost all cases, the ReLU baseline is surpassed by optimized activation functions (TanH with prefactor, learned AFs, and learned IAFs).

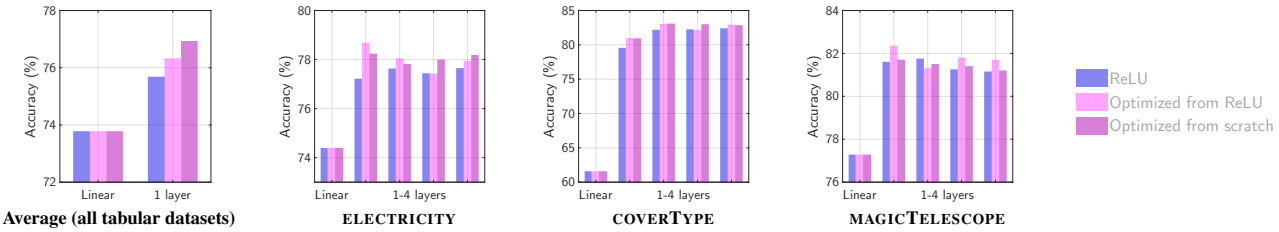

Figure 21. (Left) On the tabular datasets, the activation functions learned from scratch (initialized with zeros) are usually better than from an initialization as ReLUs. But this varies across datasets and the opposite is sometimes true (right). Models with learned activation also often perform best with fewer layers than with ReLUs, such as on the three datasets pictured.

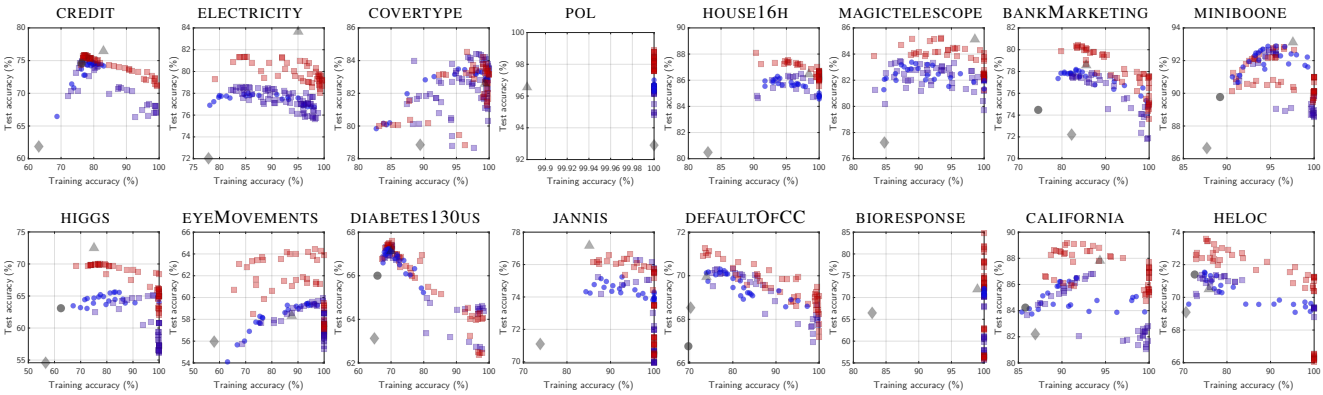

Figure 22. **Training vs. test accuracy** for all tabular datasets. The accuracy of ReLUs is generally surpassed by TanHs with the right prefactor. The accuracy is almost always best with the learned IAFs. As expected, these better models also show a clearly higher complexity. We also observe that the k-NN/trees/learned AFs models lie outside the pareto front of the ReLU models. In other words, they exhibit a different relation between training and test accuracy, which indicates that they clearly posses different inductive biases.

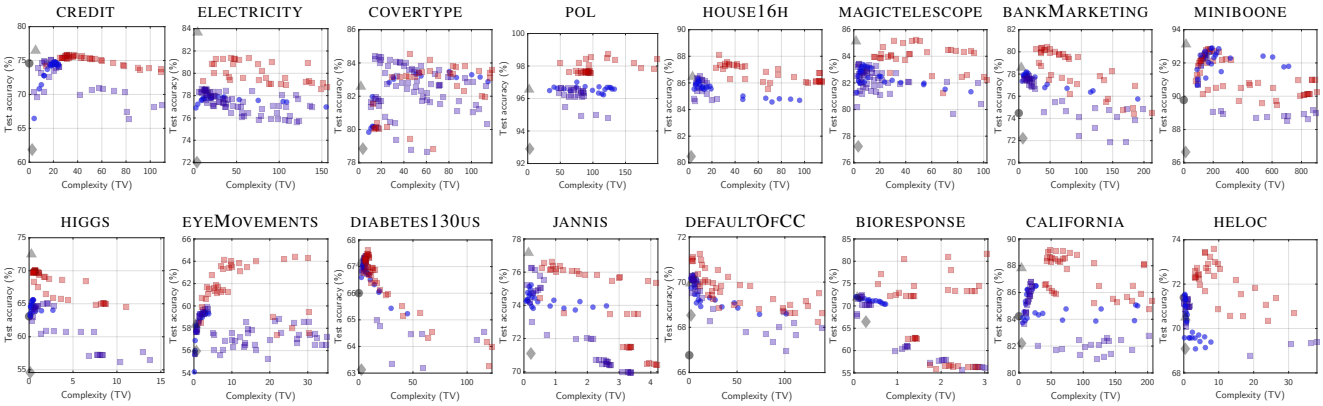

Figure 23. **Test accuracy vs. complexity** for all tabular datasets. As highlighted in Figure 6, the accuracy peaks at different complexity levels across datasets. This explains why *dataset-specific* activation functions (and inductive biases) outperform the baselines.

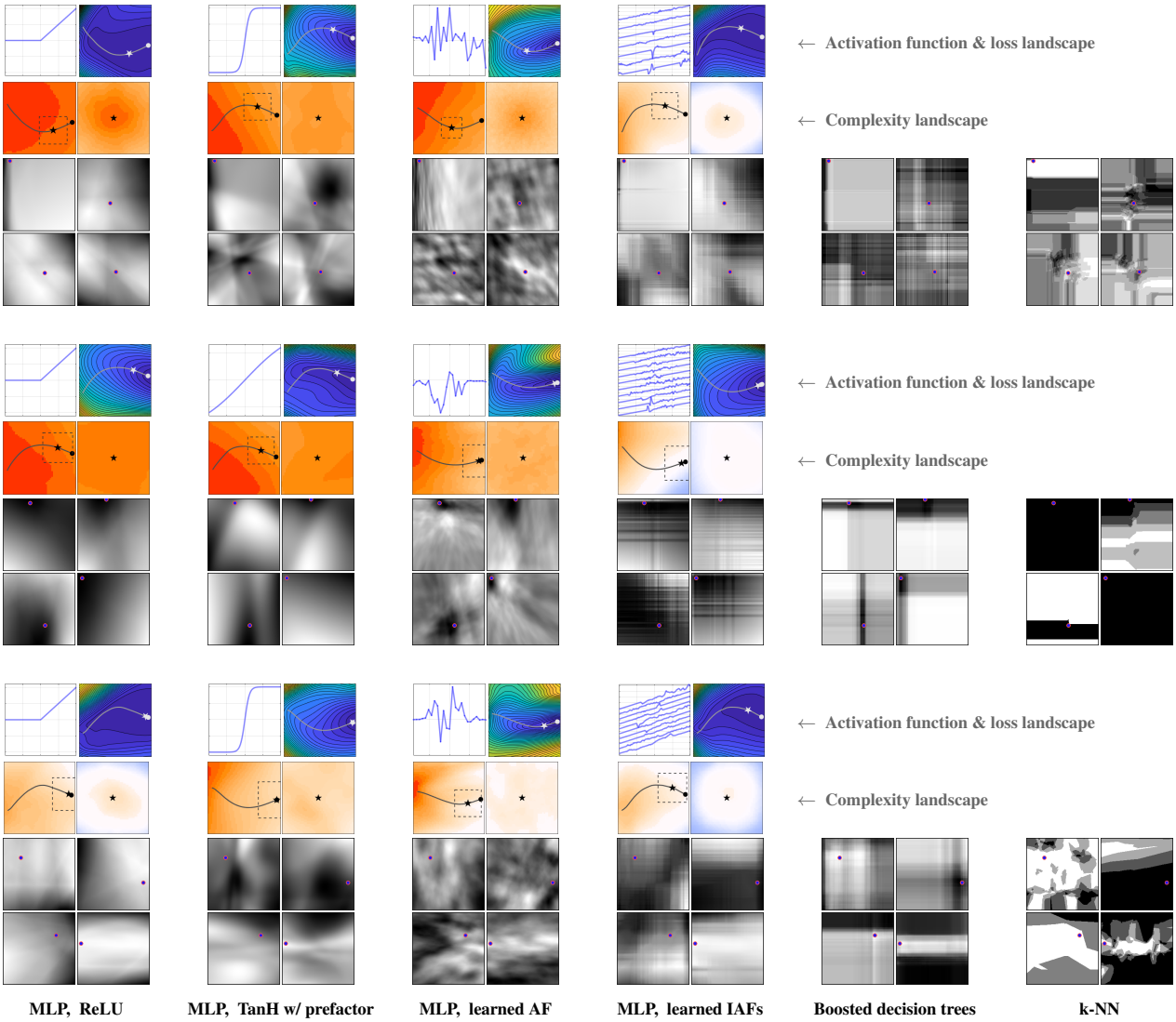

Figure 24. Models trained on three tabular datasets: ELECTRICITY MAGICTELESCOPE, and COVERTYPE [34]. See Figure 7 in the main paper for details on the meaning of these visualizations. The observations are similar across datasets.

E.5. Shortcut Learning

Data. The collages datasets are built using all 10 classes from the original datasets. This is a more difficult task that prior work [85, 93] that only used 2 classes from each dataset. Our training set uses random combinations of training images from the original dataset. Ditto for the validation and test sets. When no validation data is defined, we hold out a fraction of the training set of the same size as the test set.

Architecture. Our models are 1-layer fully-connected MLPs of width 32, trained with large-batch SGD (4096 examples per mini-batch) and a learning rate of 0.01. Only the activation function varies across experiments.

Spectral normalization. Our most successful experiments on shortcut learning use **spectral normalization** [30, 79] on all layers when training and using the learned activation functions. The motivation comes from Teney et al. [96] who showed that the magnitude of the weights in a layer, together with the choice of activation function, influences the level of "preferred complexity" of the network. We therefore hypothesized that the level of "preferred complexity" of a learned activation would be more stable (invariant to weight magnitudes) if these could be constrained in a narrow range. Spectral normalization is one way to constrain the magnitude of the linear transformation. We compare in Figure 25 the same experiments performed without and with spectral normalization. We see that the ability of the learned activations to steer the model is slightly better with spectral normalization (clearer differences in the training trajectories).

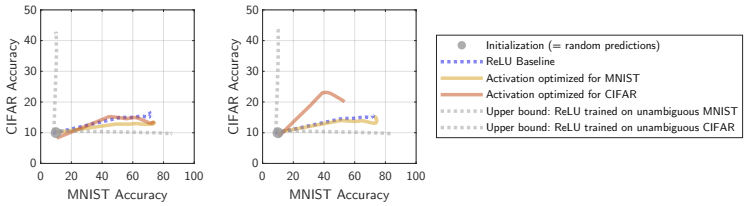

Figure 25. Experiments on shortcut learning (MNIST/CIFAR collages) **without (left)** and **with (right)** spectral normalization. The training trajectory with the activation optimized for CIFAR clearly differs from the baseline when using spectral normalization.

Additional results. We repeat our experiments with collages made from MNIST/SVHN. The training trajectories are not as distinct as with MNIST/CIFAR, but the models do also achieve different top accuracies on the two datasets.

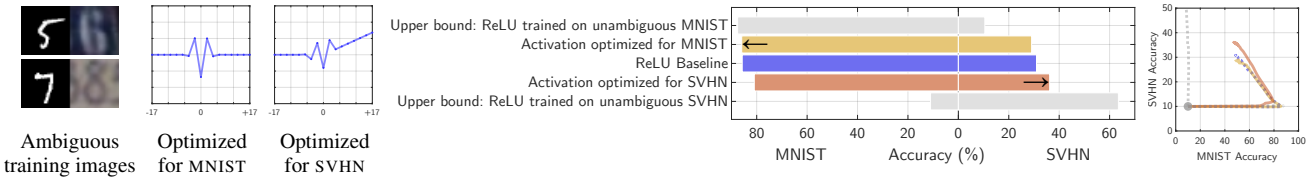

Figure 26. Experiments on shortcut learning with MNIST/SVHN collages. Similar effects are obtained as with MNIST/CIFAR (Figure 9).

E.6. Algorithmic Tasks and Grokking

Data. We visualize in Figure 10 the target functions of the algorithmic tasks used to investigate grokking used in prior work [73]. Each axis of the visualizations corresponds to one of the two discrete-valued operands. Grayscale values correspond to the target function's output, scaled to fit within the black $\rightarrow$ white range. From the point of view of a network, operands and output are represented as one-hot vectors. For example, for the task $a+b \mod 27$, each operands can take 27 different values. Each is represented by a one-hot vector of length 27. The two are concatenated such that the input to the network is a vector of size $2 \times 27 = 54$. The output of the network is a classification over the 27 possible values. For every task, we generate all possible data (i.e. every combination of values of the two operands) and make random 80/20 training/test splits.

Architecture. All networks in this section are 1-hidden layer MLPs of width 256, trained with an MSE loss and large-batch (4096) gradient descent, no weight decay, learning rate of 1.0, for max. $6e4$ training steps.

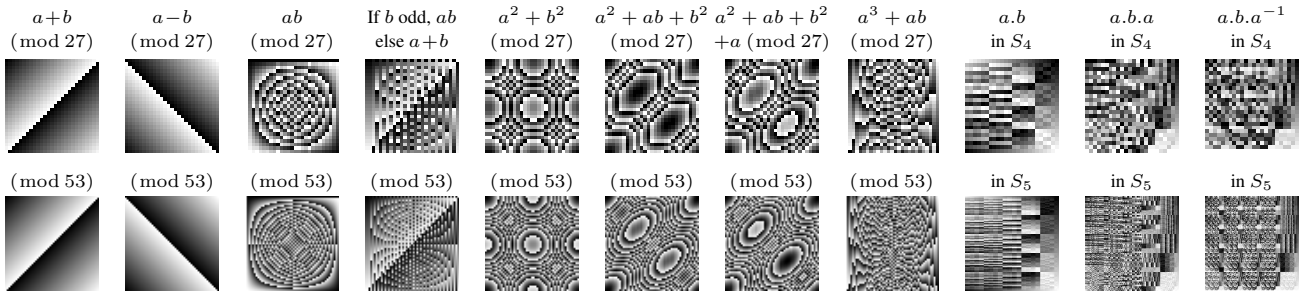

Figure 27. Algorithmic tasks used to investigate grokking, also used by Power et al. [73]. Each task is defined as an operation over two discrete-valued operands, passed to the model as one-hot encodings. We visualize the target function of each task by plotting its value over all possible values of the operands (corresponding to the X/Y axes of each image). S_n is the group of permutations of n elements ($|S_4| = 24$, $|S_5| = 120$).

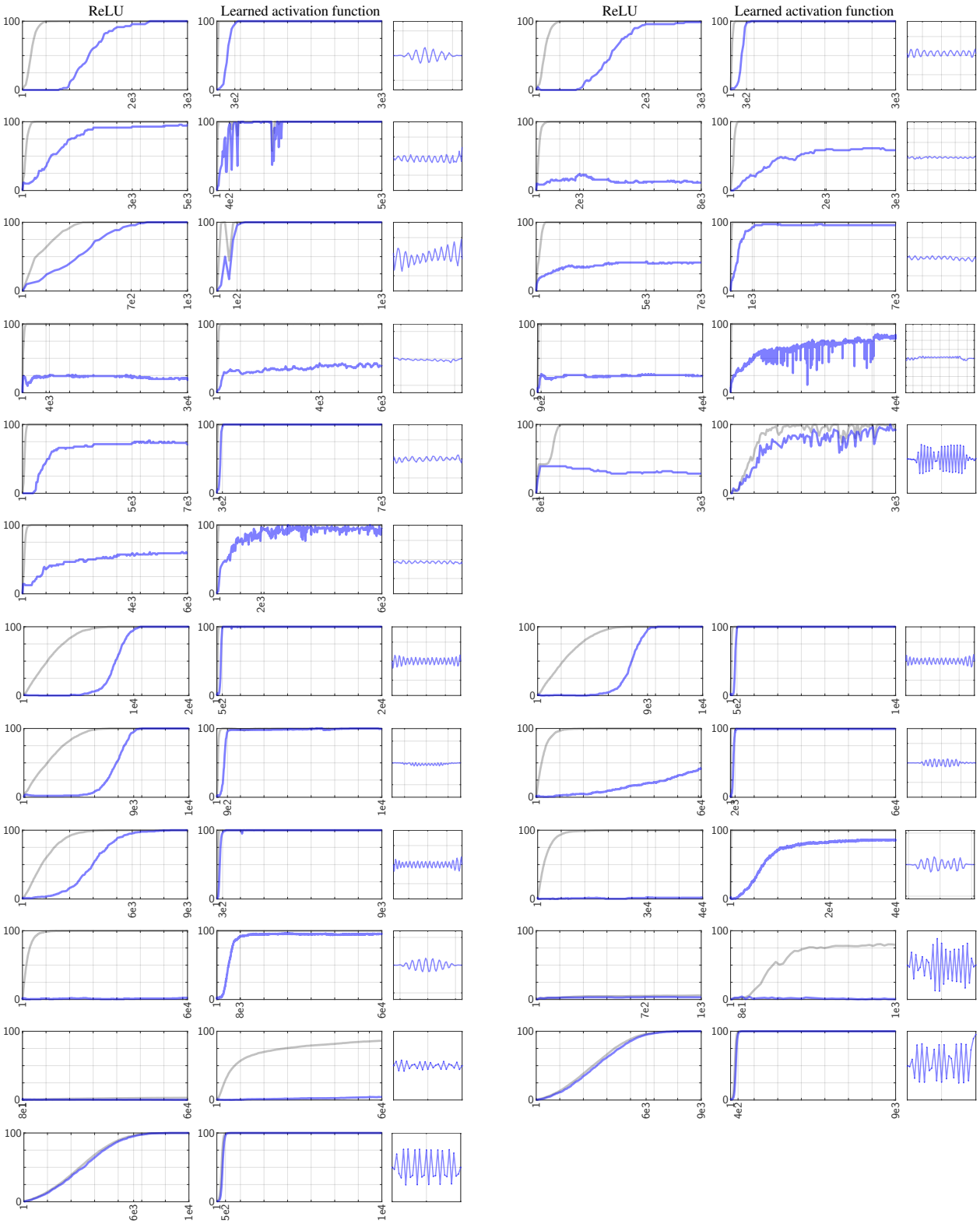

Figure 28. Training curves for all tasks from Figure 27 (same order, left-to-right then top-to-bottom). For each task, we show the accuracy across training iterations (— training accuracy, — test accuracy) for models with ReLUs and learned activations, and the learned activation function itself over $[-1, 1]$. In almost all cases, the learned activation functions converge faster and/or to a higher test accuracy than ReLUs.

F. Measure of Complexity based on Total Variation

Validation against Fourier complexity. To validate the proposed measure of complexity based on total variation (TV), we compare it against a Fourier-based measure from prior work [96]. We plot the two for a large number of models in Figure 29. They are very closely correlated. The TV is discriminative for both small and large values, its evaluation is numerically more stable, and it is more straightforward to implement. We made similar observations with other models and other datasets.

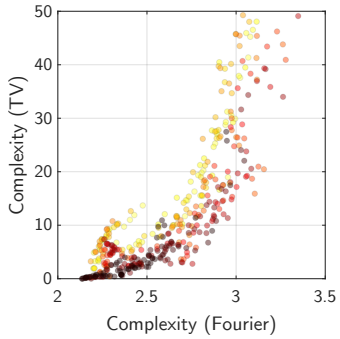

Figure 29. Comparison of the measure of complexity based on total variation (TV) vs. a Fourier-based measure of complexity from prior work [96]. We plot the values for a large number of models trained on the ELECTRICITY tabular dataset [34]. The models use a TanH activation with a prefactor ranging from 0.1 to 8. The shade of the markers corresponds to the value of the prefactor (darker $\approx$ smaller).

Implementation. The TV complexity involves a few implementation choices. Most are not critical as long as they are consistent across values being compared. We provide precise hyperparameter values that we used but they are easy to tune. One can simply evaluate the TV of some models multiple times (with different random seeds for the choice of paths) and verify that the variance is small.

- Number of linear paths: 200. This simply needs to be high enough to probe the function along many dimensions.
- Number of points on each path: 100. This simply need to be high enough to capture the resolution of the variations in the function (see Figure 30).
- The two points defining each path are chosen as two points from the training set with **different labels**. One can also include points with the same label, but the path between them often is a constant line that does not bring any information.
- We account for the fact that the function may not perfectly fit the ground truth values by first subtracting, from the evaluated path (blue line in Figure 30), the straight path connecting the predicted values at the two end points. What we measure is therefore the **deviation from a piecewise linear model**. Therefore, by design, the TV complexity of a linear model is 0.
- Conceptually, it could make sense to normalize the TV of every path by the distance between its end points, because more variations could be expected along a longer path. In practice however, this would make no difference as long as the complexity values being compared are measured on the same set of paths/end points, or even just many paths from the same distribution of end points (i.e. the same dataset).

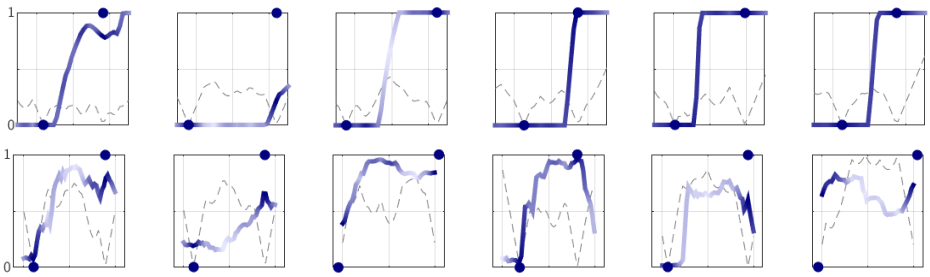

Figure 30. Examples of 1D paths (along the X axis) used to compute the TV complexity. The Y axis represents the model output. These examples correspond to a model trained on a tabular classification dataset with ground truth labels in $\{0, 1\}$. The blue dots (●) represent the paths' end points, which are training examples picked at random, and their ground truth values. The blue lines (—) represent the output of the model (capped to $[0, 1]$ for visualization). Note that this model does not perfectly interpolate the training points, i.e. the line does not always pass through the blue points. Dashed lines in the background represent the distance to the closest point in the dataset, for debugging purposes.