Neural Redshift: Random Networks are not Random Functions

Supplementary Material

A. Additional Related Work

A vast literature examines the success of deep learning using inductive biases of optimization methods (*e.g.* SGD [48]) and architectures (*e.g.* CNNs [11], transformers [90]). This paper instead examines *implicit* inductive biases in *unstructured* architectures.

Parametrization of NNs. It is challenging to understand the structure and "effective dimensionality" of the weight space of NNs because multiple weight configurations and their permutations correspond to the same function [17, 71, 89]. A recent study quantified the information needed to identify a model with good generalization [8]. However, the estimated values are astronomical (meaning that no dataset would ever be large enough to learn the target function). Our work reconciles these results with the reality (the fact that deep learning does work in practice) by showing that the overlap of the set of good generalizing functions with uniform samples in *weight space* [8, Fig. 1] is much denser than its overlap with *truly random* functions. In other words, random sampling in weight space generally yields functions likely to generalize. Much less information is needed to pick one solution among those than estimated in [8].

Some think that "*stronger inductive biases come at the cost of decreasing the universality of a model*" [13]. This is a misunderstanding of the role of inductive biases: they are fundamentally necessary for machine learning and they do not imply a restriction on the set of learnable functions. We show in particular that MLPs have strong inductive biases yet remain universal.

The simplicity bias refers to the observed tendency of NNs to fit their training data with simple functions. It is desirable when it prevents overparametrized networks from overfitting the training data [3, 53]. But it is a curse when it causes shortcut learning [25, 75]. Most papers on this topic are about trained networks, hence they confound the inductive biases of the architectures and of the optimization. Most explanations of the simplicity bias involve loss functions [52] and gradient descent [2, 32, 41, 73].

Work closer to ours [12, 44, 79] examines the simplicity bias of networks with **random weights**. These studies are limited to MLPs with binary inputs/outputs, ReLU activations, and/or simplicity measured as compressibility. In contrast, we examine more architectures and other measures of complexity. Earlier works with random-weight networks include [23, 27, 40, 43, 50, 54, 56, 63].

Goldblum et al. [29] proposed that NNs are effective because they combine a simplicity bias with a flexible hypothesis space. Thus they can represent complex functions and benefit from large datasets. Our results also support this argument.

The spectral bias [57] or frequency principle [84] is a particular form of the simplicity bias. It refers to the observation that NNs learn low-frequency components of the target function earlier during training.[6] Works on this topic are specific to gradient descent [70, 85]. and often to ReLU networks [35, 38, 88]. Our work is about properties of architectures independent of the training.

Work closer to ours [86] has noted that the spectral bias exists with ReLUs but not with sigmoidal activations, and that it depends on weight magnitudes and depth (all of which we also observe in our experiments). Their analysis uses the neural tangent kernel (NTK) whereas we use a Fourier decomposition of the learned function, which is arguably more direct and intuitive. We also examine other notions of complexity, and other architectures.

In work concurrent to ours, Abbe et al. [1] used Walsh decompositions (a variant of Fourier analysis suited to binary functions) to characterize learned binary classification networks. They also propose that typical NNs preferably fit low-degree basis functions to the training data and this explains their generalization capabilities. Their discussion, which focuses on classification tasks, is highly complementary to ours.

The "deep image prior" [78] is an image processing method that exploit the inductive biases of an untrained network. However it specifically relies on convolutional (U-Net) architectures, whose inductive biases have little to do with those studied in this paper.

Measures of complexity. Quantifying complexity is an open problem in the fundamental sciences. Algorithmic information theory (AIT) and Kolmogorov complexity are one formalization of this problem. Kolmogorov complexity has been proposed as an explicit regularizer to train NNs by Schmidhuber [62]. Dingle et al. [15] used AIT to explain the prevalence of simplicity in the real-world with examples in biology and finance. Building on this work, Valle-Perez et al. [79] showed that binary ReLU networks with random weights have a similar bias for simplicity. Our work extends

[6]**Frequencies of the target function**, used throughout this paper, should not be confused with frequencies of the input data. For example, high frequencies in images correspond to sharp edges. High frequencies in the target function correspond to frequent changes of label for similar images. A low-frequency target function means that similar inputs usually have similar labels.

this line of inquiry to continuous data, to other architectures, and to other notions of complexity.

Other measures of complexity for to machine learning models include four related notions: sensitivity, Lipschitz constant, norms of input gradients, and Dirichlet energy [14]. Hahn et al. [30] adapted "sensitivity" to the discrete nature of language data to measure the complexity of language classification tasks and of models.

Simplicity bias in transformers. Zhou et al. [90] explain generalization of transformer models on toy reasoning tasks using a transformer-specific measure of complexity. They propose that the function learned by a transformer corresponds to the shortest program (in a custom programming language) that could generate the training data. Bhattamishra et al. [7] showed that transformers are more biased for simplicity than LSTMs.

Controlling inductive biases. Recent work has investigated how to explicitly tweak the inductive biases of NNs through learning objectives [75, 76] and architectures [10, 74]. Our results confirms that the choice of **activation function** is critical [16]. Most studies on activation functions focus on individual neurons [63] or compare the generalization properties of entire networks [49]. Francazi et al. [18] showed that some activations cause a model at initialization to have non-uniform preference over classes. Simon et al. [67] showed that the behaviour of a deep MLP can be mimicked by a single-layer MLP with a specifically-crafted activation function.

Implicit neural representations (INRs) are an application of NNs with a need to control their spectral bias. An INR is a regression network trained to represent *e.g.* one specific image by mapping image coordinates to pixel intensities (they are also known as *neural fields* or *coordinate MLPs*). To represent sharp image details, a network must represent a high-frequency function, which is at odds with the low-frequency bias of typical architectures. It has been found that replacing ReLUs with periodic functions [81, Sect. 5], Gaussians [58], or wavelets [61] can shift the spectral bias towards higher frequencies [60]. Interestingly, such architectures (Fourier Neural Networks) were investigated as early as 1988 [22]. Our work shows that several findings about INRs are also relevant to general learning tasks.

B. Why Study Random-Weight Networks?

A motivation can be found in prior work that argued for interpreting the inductive biases of an architecture as a prior over functions that plays in the training of the model by gradient descent.

Mingard et al. [44] and Valle-Perez et al. [79] argued that the probability of sampling certain functions upon random sampling in parameter space could be treated as a prior over functions for Bayesian inference. They then presented preliminary empirical evidence that training with SGD does approximate Bayesian inference, such that the probability of landing on particular solutions is proportional to their prior probability when sampling random parameters.

C. Formal Statement of the NRS

We denote with

- F: the **target function** we want to learn;
- $f_{\boldsymbol{\theta}}$: a chosen **neural architecture** with parameters $\boldsymbol{\theta}$;
- $f^\star := f_{\boldsymbol{\theta}^\star}$: a **trained network** with $\boldsymbol{\theta}^\star$ optimized s.t. $f^\star$ approximates F;
- $\bar{f} := f_{\bar{\boldsymbol{\theta}}},\ \bar{\boldsymbol{\theta}} \sim p_{\text{prior}}(\boldsymbol{\theta})$: an **untrained random-weight network** with parameters drawn from an uninformed prior, such as the uniform distribution used to initialize the network prior to gradient descent.
- $\mathrm{C}(f)$: a scalar estimate the of **complexity** of the function f as proposed in Section 2;
- $\mathrm{perf}(f)$: a scalar measure of **generalization performance** *i.e.* how well f approximates F, for example the accuracy on a held-out test set.

The Neural Redshift (NRS) makes three propositions.

1. **NNs are biased to implement functions of a particular level of complexity determined by the architecture.**
2. **This preferred complexity is observable in networks with random weights from an uninformed prior.**

   Formally, $\forall$ architecture f, distribution $p_{\text{prior}}(\boldsymbol{\theta})$, $\exists$ preferred complexity $c \in \mathbb{R}$ s.t.

   $$\mathrm{C}(\bar{f}) = c \quad \text{with very high probability, and}$$
   $$\mathrm{C}(f^\star) = g(c) \text{ with } g : \mathbb{R} \to \mathbb{R} \text{ a monotonic function.}$$

   This means that the choice of architecture shifts the complexity of the learned function up or down similarly as it does an untrained model's. The precise shift is usually not predictable because $g(\cdot)$ is unknown.

3. **Generalization occurs when the preferred complexity of the architecture matches the target function's.**

   Formally, given two architectures f_1, f_2 with preferred complexities c_1, c_2, the one with a complexity closer to the target function's achieves better generalization:

   $$|\,\mathrm{C}(F) - g(c_1)| \;<\; |\,\mathrm{C}(F) - g(c_2)|$$
   $$\implies \;\mathrm{perf}(f_1^\star) \;>\; \mathrm{perf}(f_2^\star)\,.$$

   For example, ReLUs are popular because their low-complexity bias often aligns with the target function.

D. Technical Details

Activation functions. See Figure 11 for a summary of the activations used in our experiments and [16] for a survey.

Discrete network evaluation. For a given network that implements the function $f(\boldsymbol{x})$ of input $\boldsymbol{x} \in \mathbb{R}^d$, we obtain obtain a discrete representation as follows. We define a sequence of points $\mathbf{X}_{\text{grid}} = \{\boldsymbol{x}_i\}_{i=1}^{m^d}$ corresponding to a regular grid on the d-dimensional hypercube $[-1,1]^d$, with m values in each dimension ($m = 64$ in our experiments) hence m^d points in total. We evaluate the network on every point. This gives the sequence of scalars $\mathbf{Y}_f = \{f(\boldsymbol{x}_i) : \boldsymbol{x}_i \in \mathbf{X}_{\text{grid}}\}$.

Visualizations as grayscale images. For a network f with 2D inputs ($d = 2$) we produce a visualization as a grayscale image as follows. The values in $\mathbf{Y}_f$ are simply scaled and shifted to fill the range from black (0) to white (1) as:

$$\tilde{\mathbf{Y}} = (\mathbf{Y} - \min(\mathbf{Y}))\,/\,(\max(\mathbf{Y}) - \min(\mathbf{Y})).$$

We then reshape $\tilde{\mathbf{Y}}$ into an $m \times m$ square image.

Measures of complexity. We use our measures of complexity based on Fourier and polynomial decompositions only with $d = 2$ because of the computational expense. These methods first require an evaluation of the network on a discrete grid as described above ($\mathbf{Y}_f$) whose size grows exponentially in the number of dimensions d.

Xu et al. [84] proposed two approximations for Fourier analysis in higher dimensions. They were not used in our experiments but could be valuable for extensions of our work to higher-dimensional settings.

Fourier decomposition. To compute the measure of complexity $\mathrm{C}_{\text{Fourier}}(f)$, we first precompute values of f on a discrete grid $\mathbf{X}_{\text{grid}}$, yielding $\mathbf{Y}_f$ as describe above. We then perform a standard discrete Fourier decomposition with these precomputed values. We get:

$$\tilde{f}(\boldsymbol{k}) = \Sigma_{\boldsymbol{x} \in \mathbf{X}_{\text{grid}}}\, \omega^{\boldsymbol{x}^\intercal \boldsymbol{k}}\, f(\boldsymbol{x})$$

where $\omega = e^{-2\pi i/m}$ and $\boldsymbol{k} \in \mathbb{Z}^d$ are discrete frequency numbers. Per the Nyquist-Shannon theorem, with an evaluation of f on a grid of m values in each dimension, we can reliably measure the energy for frequency numbers up to $m/2$ in each dimension *i.e.* for $\boldsymbol{k} \in \mathbf{K} = [0, \dots, m/2]^d$.

The value $\tilde{f}(\boldsymbol{k})$ is a complex number that captures both the magnitude and phase of the $\boldsymbol{k}$th Fourier component. We do not care about the phase, hence our measure of complexity only uses the real magnitude $|\tilde{f}(\boldsymbol{k})|$ of each Fourier component $\boldsymbol{k}$. We then seek to summarize the distribution of these magnitudes across frequencies into a single value. We define the measure of complexity:

$$\mathrm{C}_{\text{Fourier}}(f) \;=\; \Sigma_{\boldsymbol{k} \in \mathbf{K}}\, |\tilde{f}(\boldsymbol{k})| \,.\, ||\boldsymbol{k}||_2 \;\;/\;\; \Sigma_{\boldsymbol{k} \in \mathbf{K}}\, |\tilde{f}(\boldsymbol{k})|.$$

This is the average of magnitudes, weighted each by the corresponding frequency, disregarding orientation (*e.g.* horizontal and vertical patterns in a 2D visualization of the function are treated similarly), and normalized such that magnitudes sum to 1.

See [57] for a technical discussion justifying Fourier analysis on non-periodic bounded functions.

Limitations of a scalar measure of complexity. The above definition is necessarily imperfect at summarizing the distributions of magnitudes across frequencies. For example, an f containing both low and high-frequencies could receive the same value as one containing only medium frequencies. In practice however, we use this complexity measure on random networks, and we verified empirically that the distributions of magnitudes are always unimodal. This summary statistic is therefore a reasonable choice to compare distributions.

Polynomial decomposition. As an alternative to Fourier analysis, we use decomposition in polynomial series.[7] It uses a predefined set of polynomials $P_n(x)$, $n = [0, \dots, N]$ to approximate a function $f(x)$ on the interval $x \in [-1, 1]$ as $f(x) \approx \Sigma_{c=0}^{N}\, c_n P_n(x)$. The coefficients are calculated as $c_n = 0.5\,(2n+1) \int_{-1}^{+1} f(x)\, P_n(x)\, dx$. These definitions readily extends to higher dimensions.

In a Fourier decomposition, the coefficients indicate the amount the various frequency components in f. Here, each coefficient c_n indicates the amount of a component of a certain order. In 2 dimensions ($d = 2$), we have N^2 coefficients $c_{00}, c_{01}, \dots, c_{NN}$. We define our measure of complexity:

$$\mathrm{C}_{\text{Chebyshev}}(f) \;=\; \frac{\Sigma_{n_1,n_2=0}^{N}\, |c_{n_1 n_2}| \,.\, ||\,[n_1,n_2]\,||_2}{\Sigma_{n1,n2=0}^{N}\, |c_{n1,n2}|}.$$

This definition is nearly identical to the Fourier one.

In practice, we experimented with Hermite, Legendre, and Chebyshev bases of polynomials. We found the latter to be more numerically stable. To compute the coefficients, we use trapezoidal numerical integration and the same sampling of f on $\mathbf{X}_{\text{grid}}$ as described above, and a maximum order $N = 100$. To make the evaluation of the integrals more numerically stable (especially with Legendre polynomials), we omit a border near the edges of the domain $[-1, 1]^d$. With a 64×64 grid, we omit 3 values on every side.

LZ Complexity. We use the compression-based measure of complexity described in [15, 79] as an approximation of the Kolmogorov complexity. We first evaluate f on a grid to get $\mathbf{Y}_f$ as described above. The values in $\mathbf{Y}_f$ are reals and generally unique, so we discretize them on a coarse scale of 10 values regularly spaced in the range of $\mathbf{Y}_f$ (the granularity of 10 is arbitrary can be set much higher with virtually no effect if $\mathbf{Y}_f$ is large enough). We then apply the classical Lempel–Ziv compression algorithm on the resulting number sequence. The measure of complexity $\mathrm{C}_{\text{LZ}}(f)$ is then defined as the size of the dictionary built by the compression algorithm. The LZ algorithm is sensitive to the order

[7] See *e.g.* https://www.thermopedia.com/content/918/.

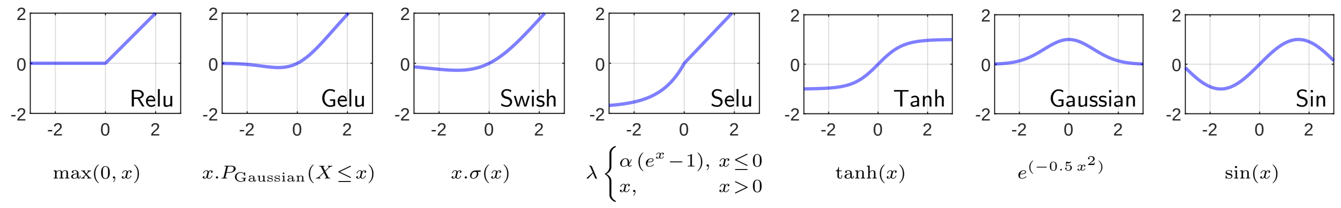


Figure 11. Activation functions used in our experiments.

of the sequence to compress, but we find very little difference across different orderings of $\mathbf{Y}_f$ (snake, zig-zag, spiral patterns). Thus we use a simple column-wise vectorization of the 2D grid.

In higher dimensions (Colored-MNIST experiments), it would be computationally too expensive to define $\mathbf{X}_{\text{grid}}$ as a dense sampling of the full hypercube $[-1, 1]^d$ (since d is large). Instead, we randomly pick m corners of the hypercube and sample m points $\boldsymbol{x}_i$ regularly between each successive pair of them. This gives a total of m^2 points corresponding to random linear traversals of the input space. Instead of feeding $\mathbf{Y}_f$ directly to the LZ algorithm, we also feed it with successive differences between successive values, which we found to improve the stability of the estimated complexity (for example, the pixel values $10, 12, 15, 18$ are turned into $2, 3, 3$).

LZ Complexity with transformers. These experiments use $\mathrm{C}_{\mathrm{LZ}}(f)$ on sequences of tokens. Each token is represented by its index in the vocabulary, and the LZ algorithm is directly applied on these sequences of integers.

Absolute complexity values. The different measures of complexity have different absolute scales and no comparable units. Therefore, for each measure, we rescale the values such that observed values fill the range $[0, 1]$.

Unbiased model. We construct an architecture that displays no bias for any frequency in a Fourier decomposition of the functions it implements. This architecture $f_{\boldsymbol{\theta}}(\cdot)$ implements an inverse discrete Fourier transform with learnable parameters $\boldsymbol{\theta} = \{\boldsymbol{\theta}_{\text{mag}}, \boldsymbol{\theta}_{\text{phase}}\}$ that correspond to the magnitude and phase of each Fourier component. It can be implement as a one-hidden-layer MLP with sine activation, fixed input weights (each channel defining the frequency of one Fourier component), learnable input biases (the phase shifts), and learnable output weights (the Fourier coefficients).

Experiments with modulo addition. These experiments use a 4-layer MLP of width 128. We train them with full-batch Adam, a learning rate 0.001, for 3k iterations with no early stopping. Each experiment is run with 5 random seeds. The Figure 7 shows the average over seeds for clarity (each point corresponds to a different architecture). Figure 8 shows all seeds (each point corresponds to a different seed).

Experiments on Colored-MNIST. The dataset is built from the MNIST digits, keeping the original separation between training and test images. To define a regression task, we turn the original classification labels $\{0, 1, \ldots, 9\}$ into values in $[0, 1]$. To introduce a spurious feature, each image is concatenated with a column of pixels of uniform grayscale intensity (the "color" of the image). This "color" is directly correlated with the label with some added noise to simulate a realistic spurious feature: in 3% of the training data, the color is replaced with a random one.

The models are 2-layer MLPs of width 64. They are trained with an MSE loss with full-batch Adam, learning rate 0.002, 10k iterations with no early stopping. The "accuracy" in our plots is actually: 1−MAE (mean average error). Since this is a regression task with test labels distributed uniformly in $[0, 1]$, this metric is indeed interpretable as a binary accuracy, with 0.5 equivalent to random chance.

Experiments with transformers. In all experiments described above, we directly examine the input $\rightarrow$ output mappings implemented by neural networks. In the experiments with transformer sequence models, we examine sequences generated by the models. These models are autoregressive, which means that the function they implement is the mapping context $\rightarrow$ next token. We expect a simple function (*e.g.* low-frequency) to produce lots of repetitions in sequences sampled auto-regressively. (language models are indeed known to often repeat themselves [21, 34]). Such sequences are highly compressible. They should therefore give a low values of C_{LZ}.

E. Additional Experiments with Trained Models

This section presents experiments with models trained with standard gradient descent. We will show that there is a correlation between the complexity of a model at initialization (*i.e.* with random weights) and that of a trained model of the same architecture.

Setup with coordinate-MLPs. The experiments in this section use models trained as implicit neural representations of images (INRs), also known as coordinate-MLPs [81]. Such a model is trained to represent a specific grayscale image, It takes as input 2D coordinates in the image plane $\boldsymbol{x} \in [-1, 1]^2$. It produces as output the scalar grayscale value of the image at this location. The ground truth data is a chosen image (Figure 12). For training, we use a subset of pixels. For testing, we evaluate the network on a 64×64 grid, which directly gives a 64×64 pixel representation of the function learned.

Why use coordinate-MLPs? This setup produces interpretable visualizations and allows comparing visually the ground truth (original image) with the learned function. Because the ground truth is defined on a regular grid (unlike most real data) it also facilitates the computation of 2D Fourier transforms. We use Fourier transforms to quantitatively compare the ground truth with the learned function and verify the third part of the NRS (generalization is enabled by matching of the architecture's preferred complexity with the target function's).

Data. We use a 64×64 pixel version of the well-known *cameraman* image (Figure 12, left) [26]. For training, we use a random 40% of the pixels. This image contains both uniform areas (low frequencies) and fine details with sharp transitions (high frequencies). We also use a synthetic *waves* image (Figure 12, right). It is the sum of two orthogonal sine waves, one twice the frequency of the other. For training, we only use pixels on local extrema of the image. They form a very sparse set of points. This makes the task severely underconstrained. A model can fit this data with a variety functions. This will reveal whether a model prefers fitting low- or high-frequency patterns.

E.1. Visualizing Inductive Biases

We first perform experiments to get a visual intuition of the inductive biases provided by different activation functions. We train 3-layer MLPs of width 64 with full-batch Adam and a learning rate of 0.02 on the cameraman and waves data. Figure 13 (next page) shows very different functions across architectures. The cameraman image contains fine details with sharp edges. Their presence in the reconstruction indicate whether the model learned high-frequency components.

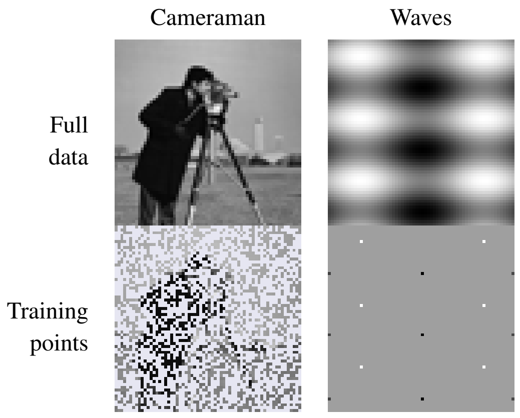


Figure 12. Data used in the coordinate-MLP experiments.

Differences across architectures. The **ReLU-like activations** are biased for simplicity, hence the learned functions tend to smooth out image details, favor large uniform regions and smooth variations. Yet, they can also represent sharp transitions, when these are necessary to fit the training data. The decision boundary with ReLUs, which is a polytope [47] is faintly discernible as criss-crossing lines in the image. Surprisingly, we observe differences across different initial weight magnitudes with ReLU, even though our experiments on random networks did not show any such effect (Section 3). We believe that this is a sign of optimization difficulties when the initial weights are large (*i.e.* difficulty of reaching a complex solution).

With **other activations** (TanH, Gaussian, sine) the bias for low or high frequencies is much more clearly modulated by the initial weight magnitude. With large magnitudes, the images contain high-frequency patterns. Similar observations are made with the waves data (Figure 14).

The **unbiased model** is useless, as expected. It reaches perfect accuracy on the training data, but the predictions on other pixels look essentially random.

With random weights. We also examine in Figure 13 the function represented by each model at initialization (with random weights). As expected, we observe a strong correlation between the amount of high frequencies at initialization and in the trained model. We also examine models at the end of training, after shuffling the trained weights (within each layer). This is another random-weight model, but its distribution of weight magnitudes matches exactly the trained model. Indeed, the shuffling preserves the weights within each layer but destroys the precise connections across layers. This enables a very interesting observation. With non-ReLU-like architectures, there is a clear increase in complexity between the functions at initialization and with shuffled weights. This means that the learned increase in **complexity in non-ReLU networks is partly encoded by changes in the distribution of weight magnitudes** (the only thing preserved through shuffling). In con-

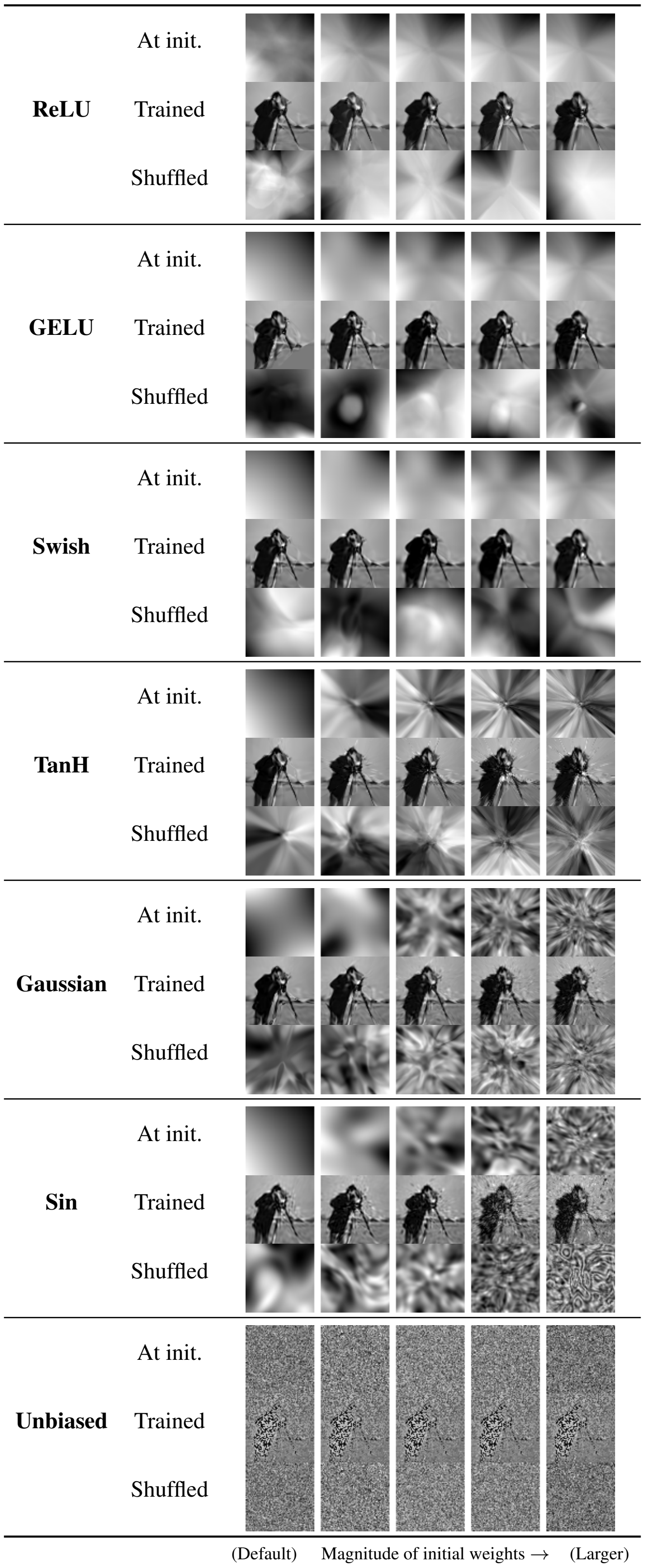


Figure 13. Coordinate-MLPs trained to represent the *cameraman* with various activations and initial weight magnitudes. The model is trained on 40% pf pixels and evaluated on a 64×64 grid. The images provide intuitions about the inductive biases of each architecture. The differences across models with random weights (at init.) and with shuffled trained weights (shuffled) show that the increase in complexity in non-ReLU models is realized by changes in weight magnitudes (which are maintained through the shuffling). In contrast, ReLU networks revert to a low complexity after shuffling, suggesting that complexity is encoded in the precise weight values, not their magnitudes.

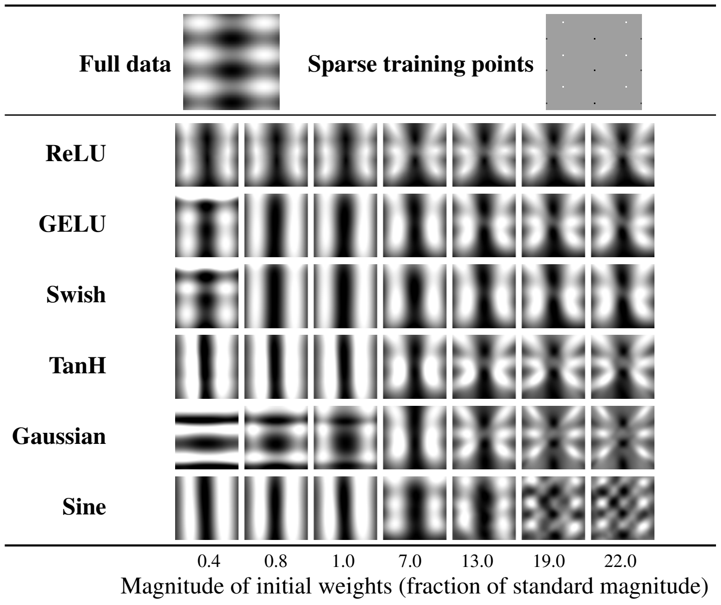


Figure 14. Coordinate-MLPs trained on sparse points of the *waves* data. Variations across learned functions show how architectures are biased towards low (ReLU) or high frequencies (Sine). ReLU activations give the most consistent behaviour across weight magnitudes.

trast, ReLU networks revert to a low complexity after shuffling. This suggests that **complexity in ReLU networks is encoded in the weights' precise values and connections across layers, not in their magnitude**.

E.2. Training Trajectories

We will now show that NNs can represent any function, but complex ones require precise weight values and connections across layers that are unlikely through random sampling but that can be found through gradient-based training.

Unlike prior work [59] that claimed that the complexity at initialization *causally* influences the solution, our results indicate instead they are two effects of a common cause (the "preferred complexity" of the architecture). The architecture is biased towards a certain complexity, and this influences both the randomly-initialized model and those found by gradient descent. There exist weight values for other functions (of complexity much lower or higher than the preferred one) but they are less likely to occur in either case.

For example, ReLU networks are biased towards simplicity but can represent complex functions. Yet, contrary to [59], initializing gradient descent with such a complex function does not yield a complex solutions after training on simple data. In other words, the architecture's bias prevails over the exact starting point of the training.

Experimental setup. We train models with different activations and initial magnitudes on the cameraman data, using $1/9$ pixels for training. We plot in Figure 16 the training trajectory of each model. Each point of a trajectory represents the average weight magnitude vs. the Fourier complexity of the function represented by the model.

Changes in weight magnitudes during training. The first observation is that the average weight magnitude changes surprisingly little. However, further examination (Figure 15) shows that the distribution shifts from uniform to long-tailed. The trained models contain more and more large-magnitude weights.

Changes in complexity during training. In Figure 16, we observe that models with ReLU-like activations at initialization have low complexity regardless of the initialization magnitude. As training progresses, the complexity increases to fit the training data. This increased complexity is encoded in the weights' precise values and connections across layers, since at the end of training, shuffling the weights reverts models back to the initial low complexity. With other activations, the initial weight magnitudes impact the complexity at initialization and of the trained model. Some of the additional complexity in the trained model seems to be partly encoded by increases in weight magnitudes, since shuffling the trained weights does seem to retain some of this additional complexity.

Summary. The choice of activation function and initial weight magnitude affect the "preferred complexity" of a model. This complexity is visible both at initialization (with random weights) and after training with gradient descent. The complexity of the learned function can depart from the "preferred level" just enough to fit the training points. Outside the interpolated training points, the shape of the learned function is very much affected by the preferred complexity.

With ReLU networks, this effect usually drives the complexity downwards (the **simplicity bias**). With other archi-

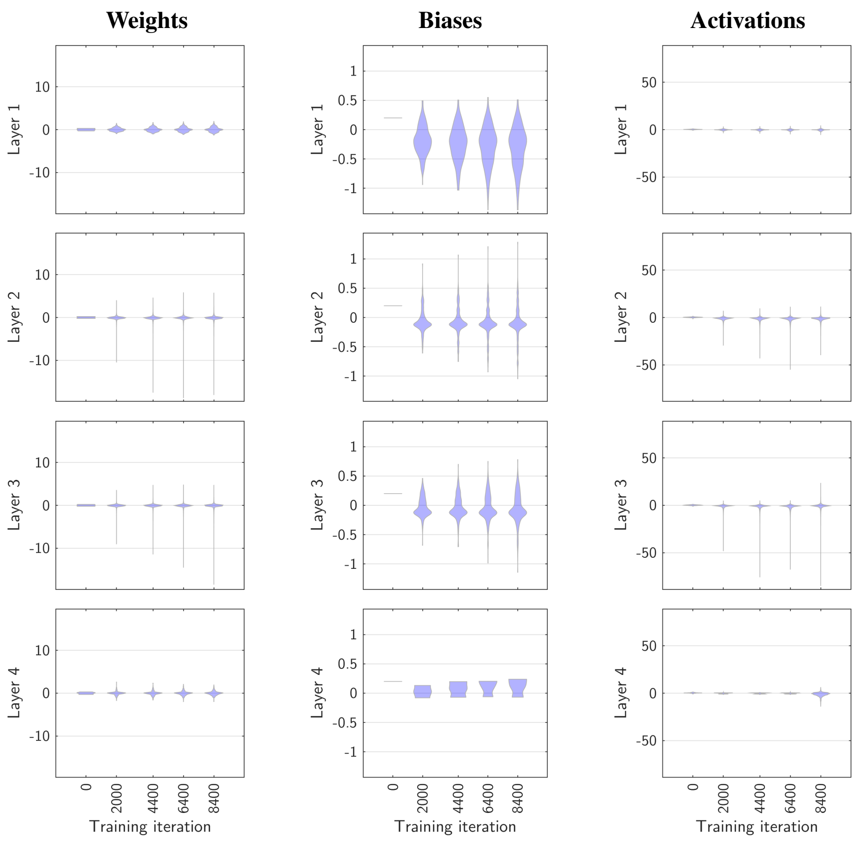


Figure 15. Distributions of the magnitudes of the weights, biases, and activations during training of a 3-layer MLP (the 4th row is the output layer) on the cameraman data. Weights and biases are initialized from a uniform distribution and zero, respectively. The distributions become very long-tailed as training progresses. The occurrence of large values is the reason why the dependence of the "preferred complexity" of certain architecture on weight magnitudes is important (it would not matter if the distribution of magnitudes remained constant throughout training).

tectures and large weight magnitudes, this often drives the complexity upwards. Both can be useful: Section 4 showed that sine activations can enable learning the parity function from sparse training points, and reduce shortcut learning by shifting the preferred complexity upwards.

Our observations also explain why the coordinate-MLPs with sine activations proposed in [68] (SIREN) require a careful initialization. This adjusts the preferred complexity to the typical frequencies found in natural images.

E.3. Pretraining and Fine-tuning

We outline preliminary results from additional experiments.

Why study fine-tuning? We have seen that the preferred complexity of an architecture can be observed with random weights. The model can then be trained by gradient descent to represent data with a different level of complexity. For example, a ReLU network, initially biased for simplicity, can represent a complex function after training on complex data. Gradient descent finely adjusts the weights to represent a complex function. We will now see how pretraining then fine-tuning on data with different levels of complexity "blends" the two in the final fine-tuned model.

Experimental setup. We pretrain an MLP with ReLU or TanH activations on high-frequency data (high-frequency 2D sine waves). We then fine-tune it on lower-frequency 2D sine waves of a different random orientation.

Observations. During early fine-tuning iterations, TanH models retain a high-frequency bias much more than ReLU models. This agrees with the proposition in E.1 that the former encode high frequencies in weight magnitudes, while ReLU models encode them in precise weight values, which are quickly lost during fine-tuning.

We further verify this explanation by shuffling the pretrained weights (within each layer) before fine-tuning. The ReLU models then show no high-frequency bias at all (since the precise arrangement of weights is completely lost through the shuffling). TanH models, however, do still show high-frequency components in the fine-tuned solution. This confirms that TanH models encode high frequencies partly in weight magnitudes since this is the only property pre-

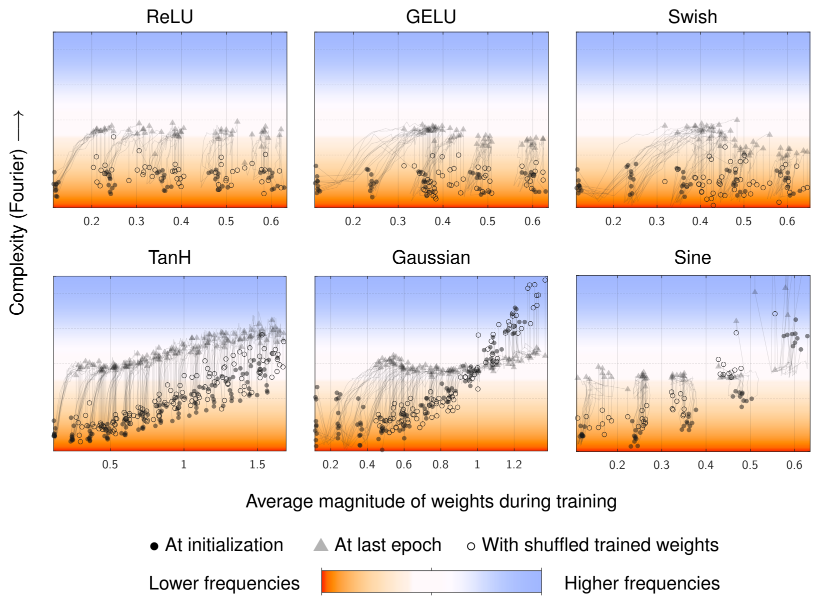


Figure 16. Training trajectories of MLP models trained on the cameraman data. Each line corresponds to one training run (with a different seed or initial weight magnitude). With ReLU-like activations, the models at initialization have low complexity regardless of the initialization magnitude. As training progresses, the complexity increases to fit the training data. This increased complexity is encoded in the weights precise values and connections across layers, since at the end of training, shuffling the weights reverts models to the initial low complexity. With other activations, the initial weight magnitude impacts the complexity at initialization and of the trained model. Some of the additional complexity in the trained model seems to be partly encoded by increases in the weight magnitudes, since shuffling the trained weights does seem to retain some of this additional complexity.

served by the shuffling.

Finally, we do not find evidence for the prior claim [59] that complexity at initialization persists *indefinitely* throughout fine-tuning. Instead, with enough iterations of fine-tuning, any pretraining effect on the preferred complexity eventually vanishes. For example, a ReLU model pretrained on high frequencies initially contains high-frequency components in the fine-tuned model. But with enough iterations, they eventually disappear *i.e.* the simplicity bias of ReLUs eventually takes over. We believe that the experiments in [59] were simply not run for long enough. This observation also disproves the causal link proposed in [59] between the complexity at initialization and in the trained model.

F. Full Results with Random Networks

On the next pages (Figures 19–21), we present heatmaps showing the average complexity of functions implemented by neural networks of various architectures with random weights and biases. Each heatmap corresponds to one architecture with varying number of layers (heatmap columns) and weight magnitudes (heatmap rows). For every other cell of a heatmap, we visualize, as a 2D grayscale image, a function implemented by one such a network with 2D input and scalar output.

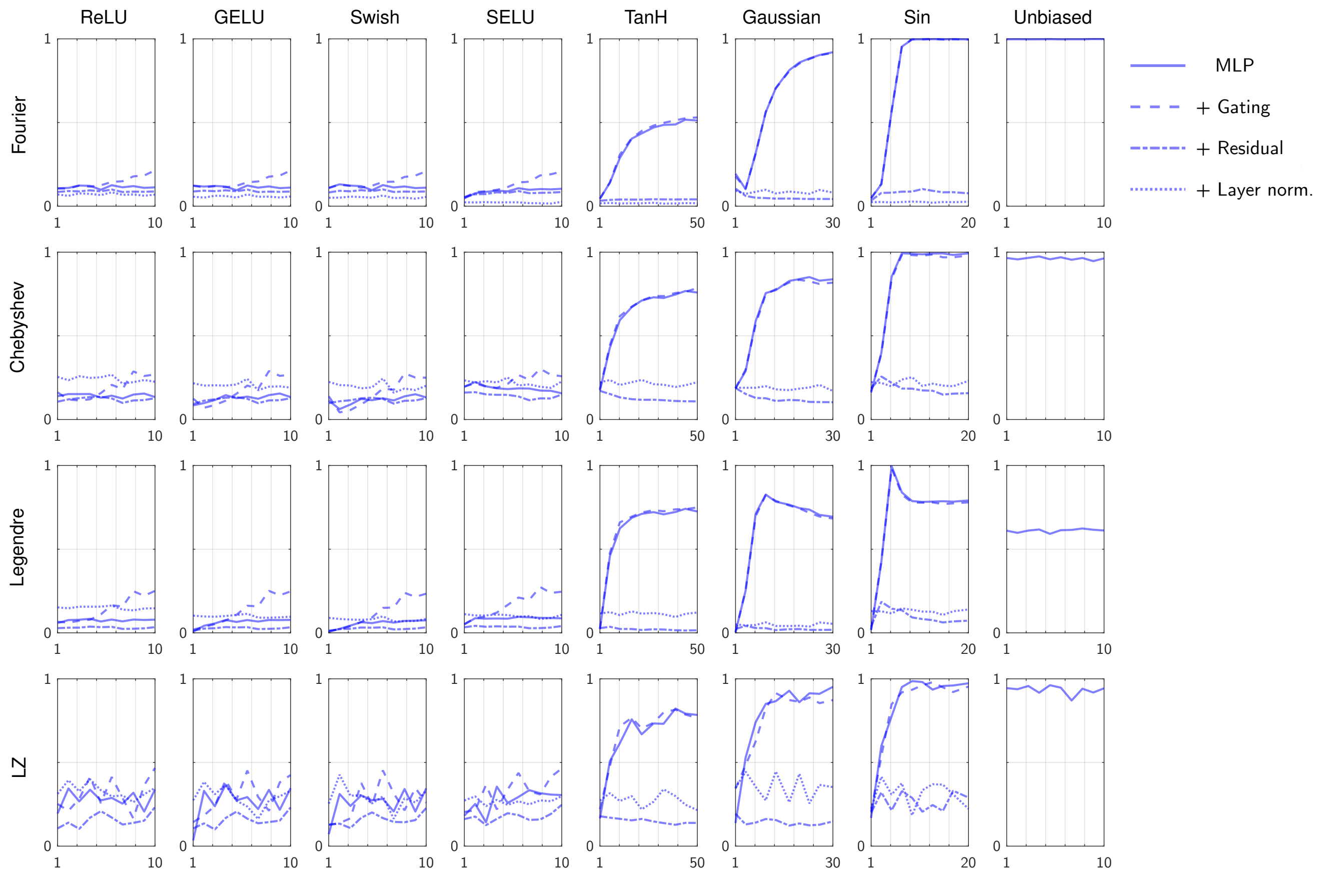


Figure 17. All **measures of complexity (Y axes)** of random networks generally increase with **weight/activation magnitudes (X axis)**. The sensitivity is however very different across activation functions (columns). This sensitivity also increases with multiplicative interactions (*i.e.* gating), decreases with residual connections, and is essentially absent with layer normalization. These effects are also visible on the heatmaps (see next pages), but faint hence visualized here as line plots.

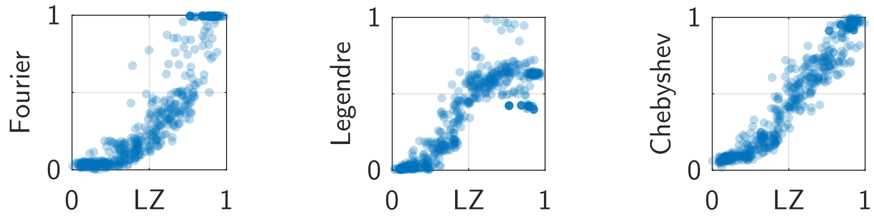


Figure 18. Correlations between our measures of complexity on random networks. They are based on frequency (Fourier), polynomial order (Legendre, Chebyshev), or compressibility (LZ). They capture different notions, yet they are closely correlated.

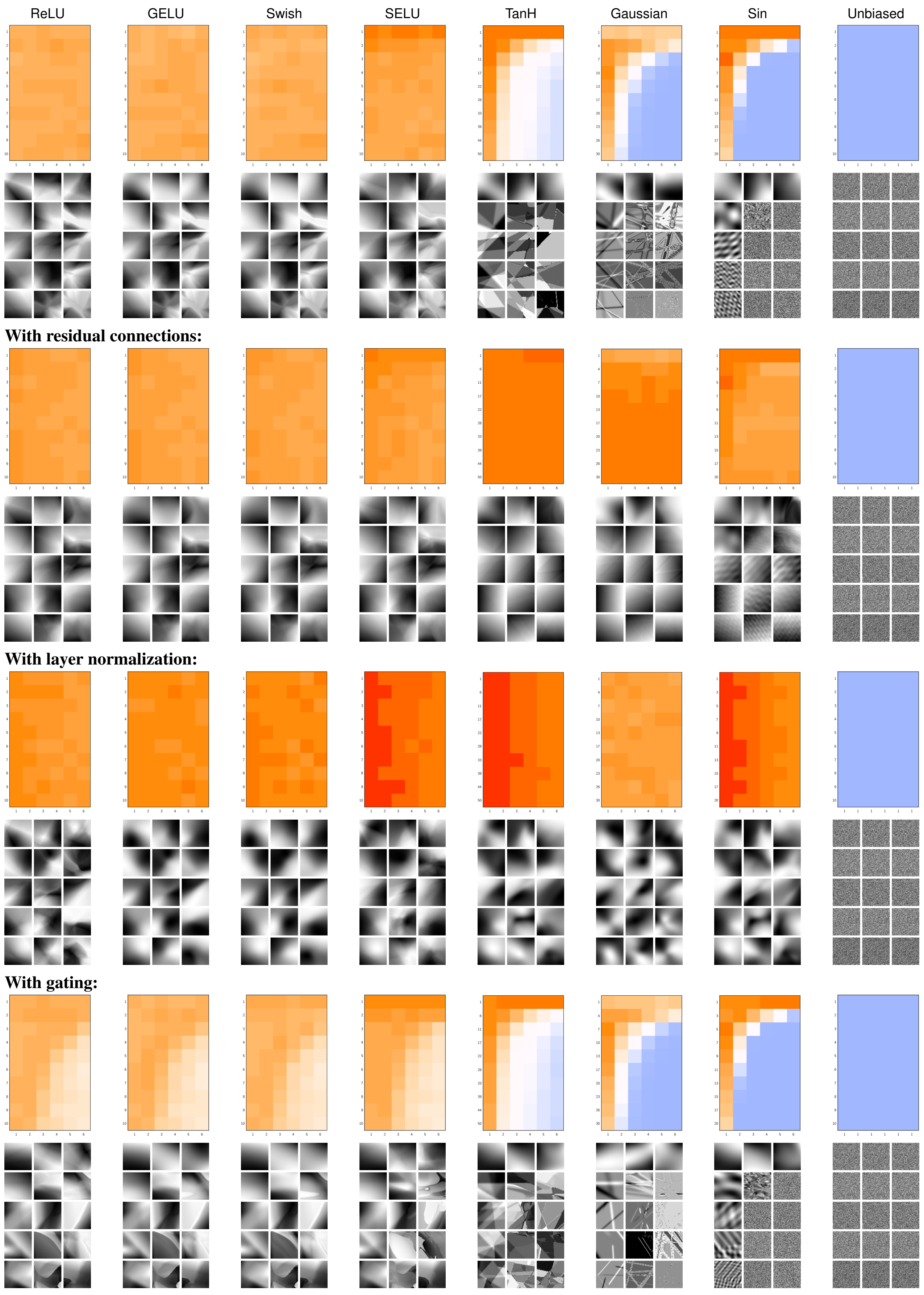
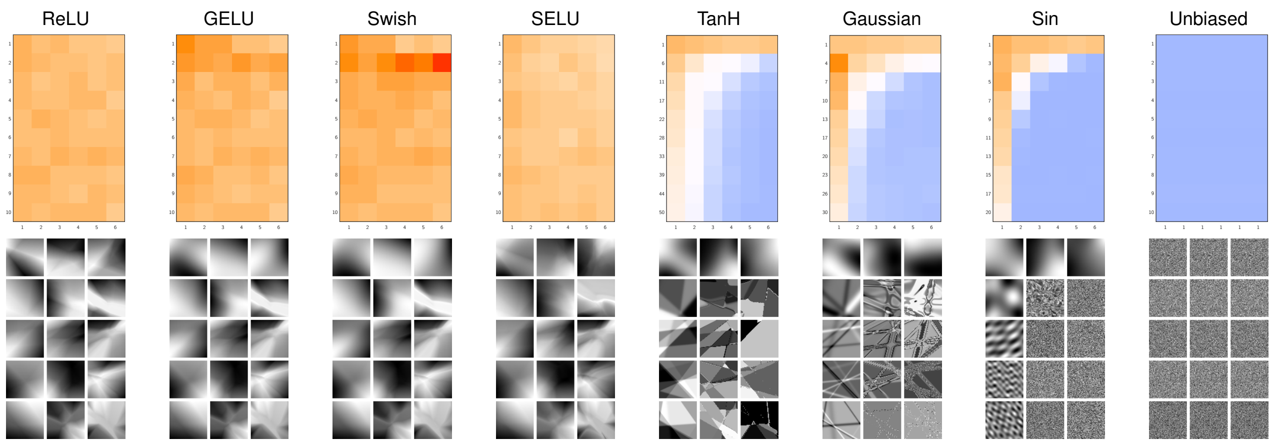
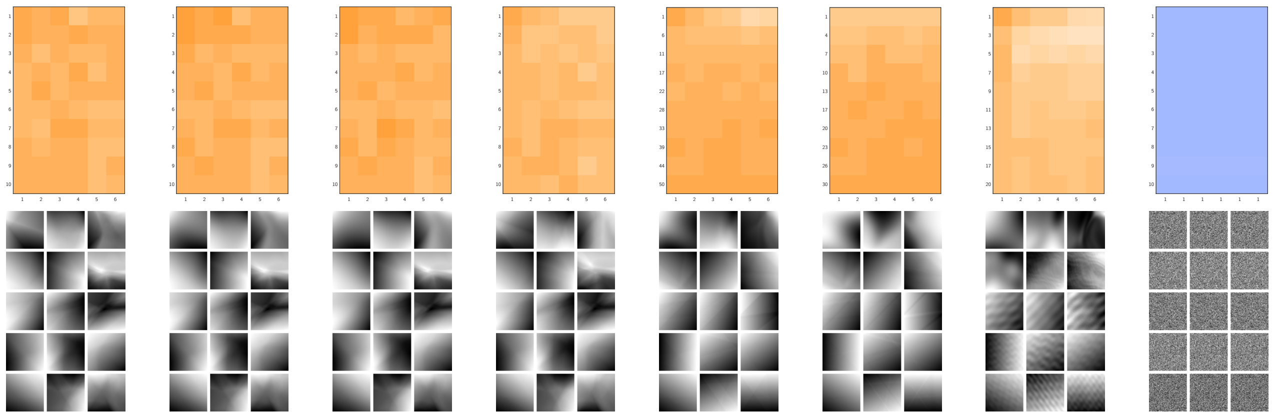


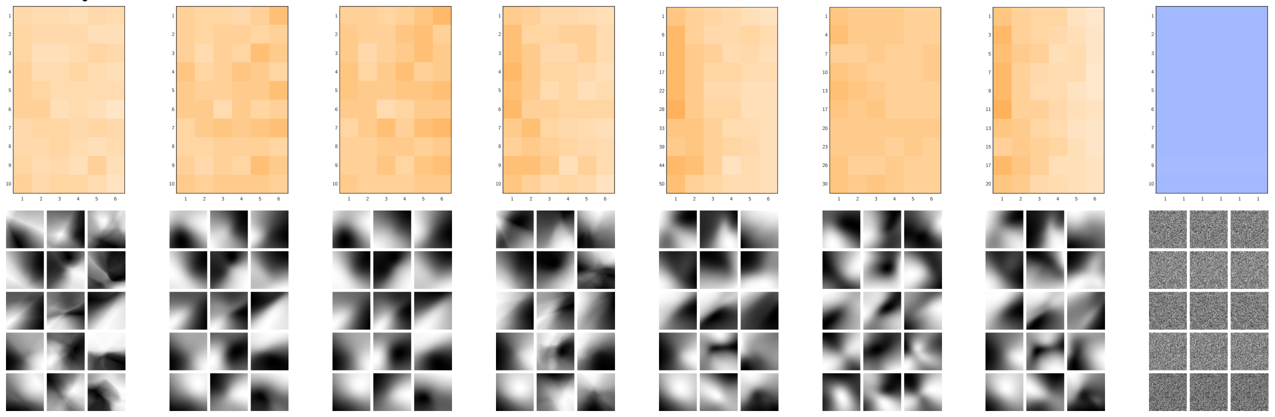
Figure 19. Heatmaps of the average complexity (**Fourier**) of various architectures with random weights, and example functions.



With residual connections:



With layer normalization:



With gating:

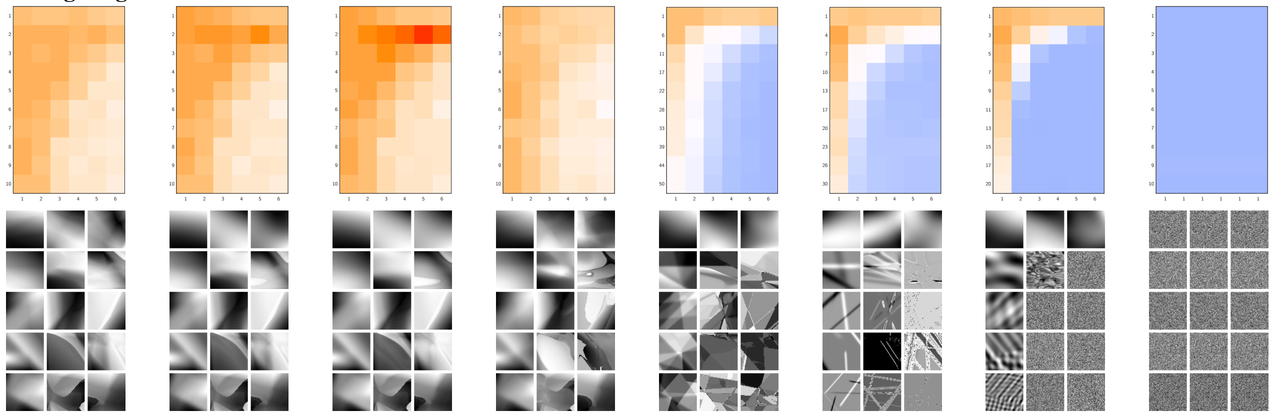


Figure 20. Same as Figure 19 with the LZ measure instead of the Fourier-based one. Results are nearly identical.

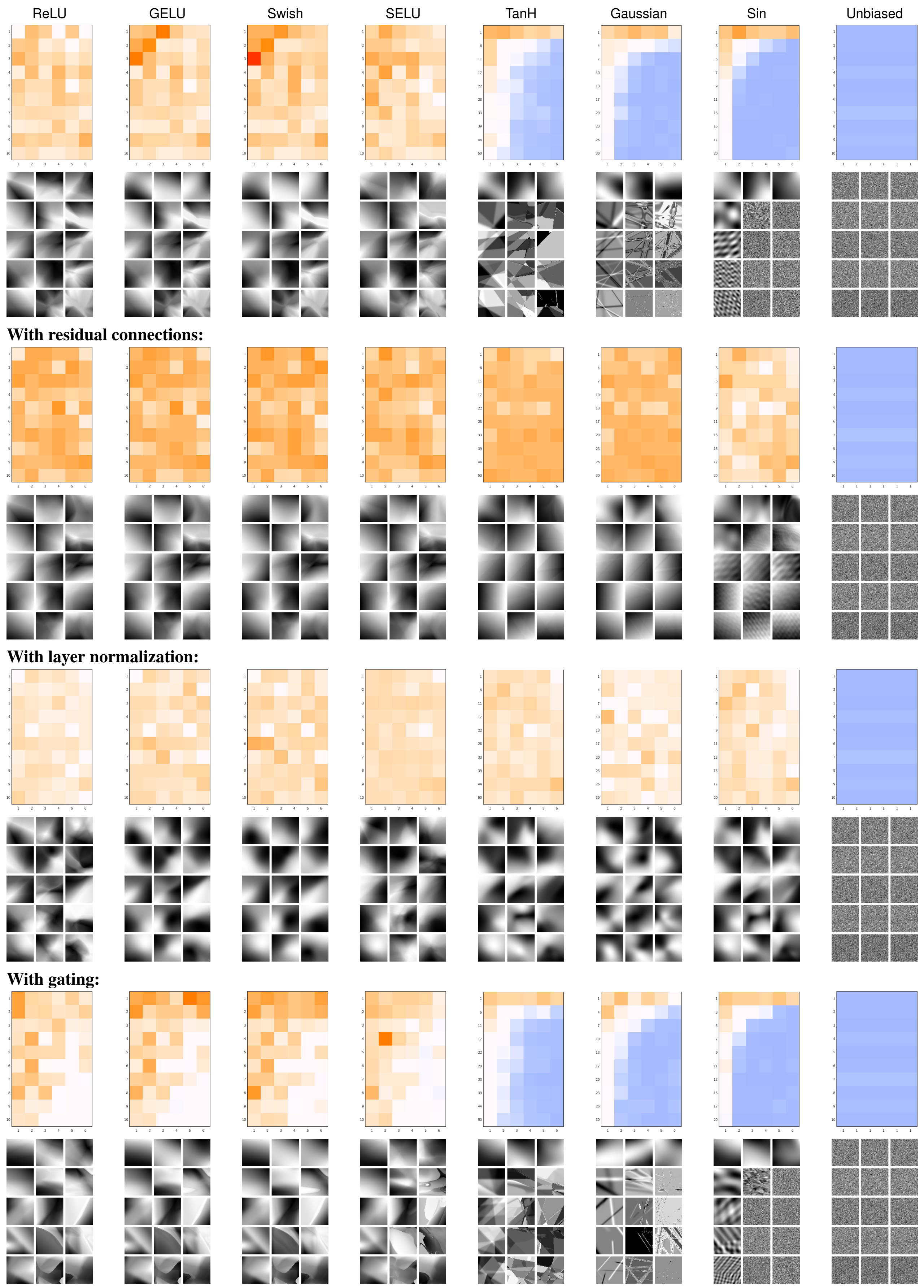


Figure 21. Same as Figure 19 with the LZ measure instead of the Fourier-based one. Results are nearly identical but noisier.