Algorithm 1 `sample-svs` (N, α): Sample singular values.

Input: $N \geq 2$, the size of the system; and $0 \leq \alpha \leq 3N$, the target σ-factor of the system.

Output: Σ, the $N \times 2$ array of singular values, satisfying $0 \leq \Sigma_{k,2} \leq \Sigma_{k,1} \leq 1$ $(\forall\ k = 1, \ldots, N)$ and $\sum_{i=1}^{N}(\Sigma_{i,1} + 2\Sigma_{i,2}) = \alpha$.

Initialize: $\Sigma \leftarrow \mathbf{0}^{N\times 2}$, the array of singular values
Initialize: $b_l \leftarrow \alpha - 3N + 3$, sampling lower bound
Initialize: $b_u \leftarrow \alpha$, sampling upper bound
for $k = 1$ to $N - 1$ **do**
  Sample $\sigma_{k,1} \sim U(\max(0, \frac{1}{3}b_l), \min(1, b_u))$
  Update $b_l \leftarrow b_l - \sigma_{k,1}$ and $b_u \leftarrow b_u - \sigma_{k,1}$
  Sample $\sigma_{k,2} \sim U(\max(0, \frac{1}{2}b_l), \min(\sigma_{k,1}, \frac{1}{2}b_u))$
  Update $b_l \leftarrow b_l - 2\sigma_{k,2} + 3$ and $b_u \leftarrow b_u - 2\sigma_{k,2}$
  Update $\Sigma_{k,1} \leftarrow \sigma_{k,1}$ and $\Sigma_{k,2} \leftarrow \sigma_{k,2}$
end for
{Note the use of b_u in both places below}
Sample $\sigma_{N,2} \sim U(\max(0, \frac{1}{2}(b_u - 1)), \frac{1}{3}b_u)$
Set $\sigma_{N,1} \leftarrow b_u - 2\sigma_{N,2}$
Update $\Sigma_{N,1} \leftarrow \sigma_{N,1}$ and $\Sigma_{N,2} \leftarrow \sigma_{N,2}$
return Σ

Algorithm 2 `sample-system` (N, b): Sample a system composed of N 2D affine transforms $\{(A_k, \mathbf{b}_k) : k = 1, \ldots, N\}$.

Input: $N \geq 2$, the size of the system; and b, a bound on the values of $\mathbf{b}_k$ such that $-b \leq \mathbf{b}_{k,i} \leq b$

Output: A set of N affine transformation parameters $(A_k, \mathbf{b}_k)$

Initialize: $S \leftarrow \{\}$, empty set of transforms
Sample $\alpha \sim U(\frac{1}{2}(5+N), \frac{1}{2}(6+N))$
$\Sigma \leftarrow$ `sample-svs` (N, α), $N \times 2$ array of singular values
for $k = 1$ to N **do**
  Sample $\theta_k, \phi_k \sim U(-\pi, \pi)$
  Sample $d_{k,1}, d_{k,2} \sim U(\{-1, 1\})$
  Sample $b_{k,1}, b_{k,2} \sim U(-b, b)$
  $R_{\theta_k} \leftarrow \begin{bmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{bmatrix}$
  $R_{\phi_k} \leftarrow \begin{bmatrix} \cos\phi_k & -\sin\phi_k \\ \sin\phi_k & \cos\phi_k \end{bmatrix}$
  $A_k \leftarrow R_{\theta_k} \begin{bmatrix} \Sigma_{k,1} & 0 \\ 0 & \Sigma_{k,2} \end{bmatrix} R_{\phi_k} \begin{bmatrix} d_{k,1} & 0 \\ 0 & d_{k,2} \end{bmatrix}$
  $\mathbf{b}_k \leftarrow \begin{bmatrix} b_{k,1} \\ b_{k,2} \end{bmatrix}$
  Insert $(A_k, \mathbf{b}_k)$ into S
end for
return S

Appendix

A. Algorithms

Described in the main paper in Section 3.2.2, we here provide precise descriptions for the `sample-svs` and `sample-system` algorithms, respectively in Algs. 1 and 2. We use these algorithms in our experiments to sample the IFS codes used in our fractal dataset.

B. Fractal Pre-training Images

Here we provide additional details on the proposed fractal pre-training images, including details on how the images are rendered as well as our procedures for "just-in-time" (on-the-fly) image generation during training.

B.1. Rendering Details

In order to add additional diversity to the rendered fractal images—to encourage the neural network to learn better, more robust representations—we supplement the rendering process (described in Section 3.1) in three ways. First, we follow the example of [15] and apply patch-based rendering, which was shown to perform better than simple point rendering. Second, we color the points on the fractal instead of rendering them as grayscale. And third, we add randomly generated backgrounds. Fig. 4 shows an example rendered fractal image with these properties (far right).

Patch-based Rendering Instead of mapping each point in $\hat{\mathcal{A}}$ to a single pixel, we follow the approach taken in [15] and map each point to a patch centered on that pixel. For each image, a patch is sampled uniformly from the set of 3×3 binary patches $\{0, 1\}^{3\times 3}$. This patch is applied for each point in $\hat{\mathcal{A}}$.

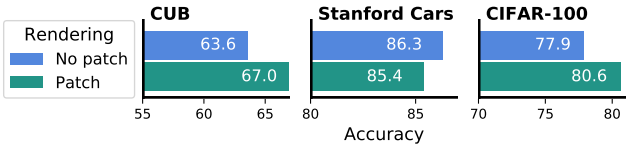

Figure 9. Fine-tuning results using models pre-trained with or without patch-based rendering.

Note: Kataoka *et al*. [15] found that patch rendering provided a fairly significant performance boost to fine-tuning. We trained a model without patch-based rendering in order to validate their findings—the results are shown in Figure 9. Our findings are consistent with [15], although for Stanford Cars the results were slightly better without patch-rendering for some reason.

Colored Fractals We adopt a simple approach for randomly coloring a fractal. First, we render the fractal in grayscale, using density-based rendering (instead of binary). Then we choose a random reference hue value, h, and assign a hue to each pixel by treating its (normalized density) grayscale value as an offset from h. We randomly sample saturation $s \sim U(0.3, 1)$ and value $v \sim U(0.5, 1)$ and apply them globally to each pixel to get an HSV image X^{hsv}, where the color for pixel i is set to be $X_i^{hsv} =$

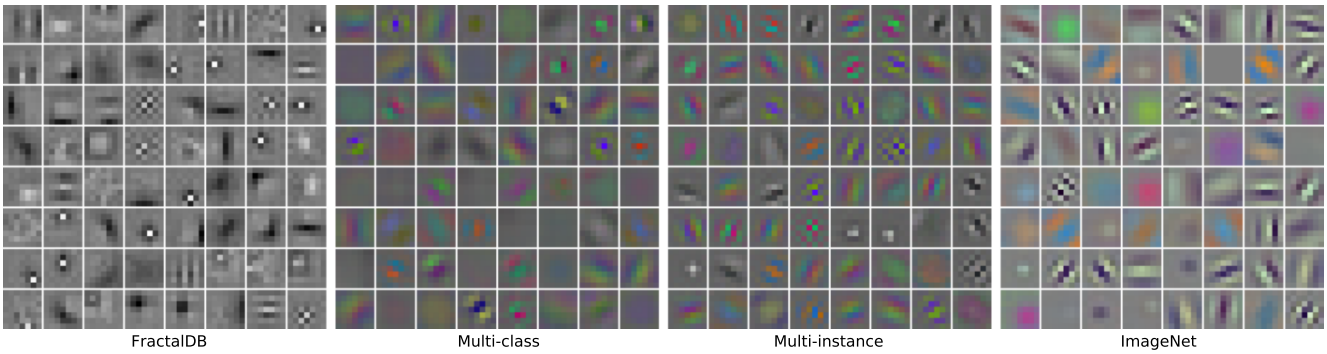

Figure 10. First layer filters learned by different pre-training methods.

$((h + X_i) \pmod{256}, s, v)$. We then convert X^{hsv} to its RGB representation X^{rgb}.

Random Backgrounds Adding backgrounds to the fractal images increases the diversity of images, and should cause the neural network model to learn to ignore backgrounds when making classification decisions. We use the midpoint-displacement, or "diamond-square" algorithm [11], to efficiently generate background textures. A parameter γ controls the roughness of the resulting texture. To generate a background, we first sample $\gamma \sim U(0.4, 0.8)$ and generate a grayscale texture image using the diamond-square algorithm. Then we colorize the texture using a process similar to the one previously described for colorizing the fractals. The final image is formed by compositing the colored fractal image on top of the random background.

B.2. Just-In-Time Image Generation

With the correct procedure, we are able to *generate* all images "on the fly" as they're needed for training. This is significant, as we circumvent the typical need to store or transmit a huge quantity of data. The entire dataset can be generated from the set of IFS codes, which can be stored in tens or hundreds of megabytes (depending on the number and size of the systems). For context, the ILSVRC2012 subset of ImageNet that is typically used for pre-training comprises 1.281M images and occupies 150GB of disk space. While in practice, we use dozens of systems per class and their augmentations (approximately 7.2MB for 1000 classes), even if 1.28M images were stored systematically as unique IFS parameters on disk, that only occupies 184.5MB, an $800\times$ reduction in storage.

Three things are necessary in order for image generation to keep up with model throughput: the first is compute-efficient fractal images; the second is efficient code; and the third is retaining a cache of recently-computed objects. Affine Iterated Function Systems are computationally efficient—a good approximation of the attractor can be achieved with a few tens or hundreds of thousands of iterations, and don't require any operations beyond basic arithmetic. We are able to get highly-efficient code by carefully writing our algorithms and compiling them with Numba [22].

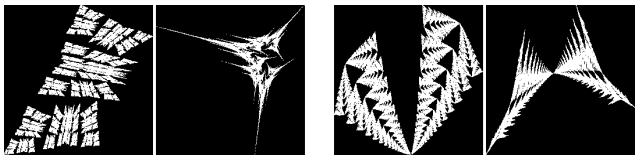

Figure 11. Examples of the effect that small perturbations in parameters can have on the resulting fractal images. In each of the two examples shown, the value of a single parameter in the IFS code was shifted by 0.1.

Even with fast code and efficient fractals, it may not be possible to generate images fast enough to match model throughput, particularly when training on multiple GPUs. As a solution, we keep a cache of recently-computed fractal images, which gets updated on a fixed schedule. For example, when training a multi-class classification model, we keep a cache of the last 512 generated images. Half of each training batch consists of images drawn from the cache and augmented using standard data augmentation practices. The other half of the batch consists of newly-generated images, which are then used to update the cache. This cuts in half the number of images that need to be generated from scratch at each iteration of training, greatly easing the computational load. Using a cache is even more critical when generating multi-instance images, as we describe in Section 4.2.2.

Note: Our target in this work is to generate images fast enough to keep up with training a ResNet50 model using distributed training on a workstation with 8 GPUs. Different hardware setups and different models may require adjustments—such as different cache sizes or update intervals—but with proper tuning the approach should work in a wide variety of circumstances.

	N						
	2	3	4	5	6	7	8
Sample-svs	11.7 ± 0.49	17.3 ± 0.76	22.4 ± 0.28	28.2 ± 0.38	33.1 ± 0.17	38.3 ± 0.48	43.3 ± 0.43
Sample-system	42.8 ± 1.04	49.2 ± 0.75	55.4 ± 0.25	60.5 ± 0.40	67.0 ± 0.43	72.7 ± 0.55	80.7 ± 2.86

Table 1. Average time (in microseconds) for sampling IFS codes of different size (N), using our Python implementation. The first row shows times for sampling singular values alone, and the second row shows times for sampling the full system (including sampling singular values).

Operation	Time (ms)
Iterate (10^5)	4.39 ± 0.034
Render (256×256)	1.46 ± 0.017
Colorize	0.23 ± 0.002
Background (256×256)	0.77 ± 0.001

Table 2. Average time (in milliseconds) for various stages of the fractal image rendering process, using our implementation (Python and Numba [22]). (**Iterate**) produces coordinates on the attractor through random iteration (100,000 iterations); (**Render**) maps the coordinates to a 256×256 grayscale image using patch-based rendering; (**Colorize**) converts the grayscale image to a color image; (**Background**) renders a random background. See B.1 for details.

C. Computational Requirements

Fractal Sampling and Rendering For reference, we report compute time for sampling systems and rendering fractal images. Compute time was measured using an Intel Xeon E3-1245 3.7GHz CPU. In Table B.2, we report the average time for sampling IFS codes for systems of size 2 up to size 8, along with the time for sampling just the singular values. In Table 2, we report the average time required for various stages of the image rendering process. The memory requirements for rendering a single image are low, requiring little more than the size of the output array.

Training In Table 3, we report training time under two different hardware settings. The first is a single node with 8 1080-Ti GPUs and 48 CPU cores. The second is two nodes, each with 4 Tesla P100 GPUs and a total of 56 CPU cores. We report the time required to train a model for 90 epochs, or 90,000,000 iterations (1,000,000 images per epoch, comparable to ILSVRC 2012), for both single-instance multi-class classification and multi-instance prediction.

D. Additional Data Examples

D.1. Small Changes to Parameters

Section 4.1.1 pointed out that small perturbations to IFS codes can sometimes result in large visual differences in the corresponding fractal images. We show examples of this in Figure 11.

Task	1×8 1080-Ti	2×4 P100
Multi-class	23h (15.3m)	18h (12m)
Multi-instance	25h (16.6m)	19.5h (13m)

Table 3. Representative pre-training times for both multi-class classification and multi-instance prediction, for two different hardware stacks: one node with 8 1080-Ti GPUs, and two nodes with 4 P100 GPUs each. The time in hours to train for 90 epochs is shown, with the approximate per-epoch training time (in minutes) shown in parentheses.

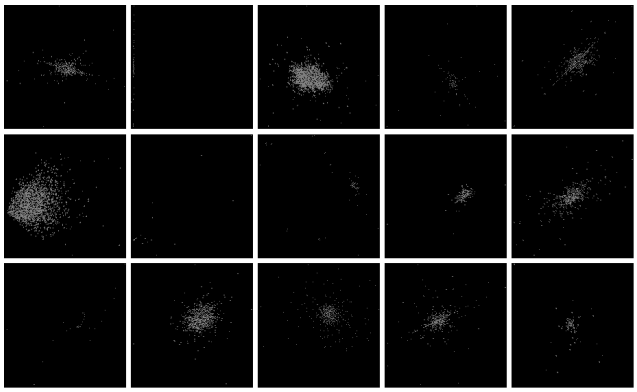

Figure 12. Examples of degenerate FractalDB images, caused by IFS parameter augmentation leading to non-contractive systems.

D.2. Problems in FractalDB

Since the data augmentation process used for FractalDB [15] doesn't enforce contractivity in the resulting IFS codes, some of the resulting images are degenerate. Figure 12 shows some sample images from the FractalDB dataset that exhibit this degeneracy, leading to small clouds of points or mostly empty images.

D.3. Example Images

Figure 16 shows images of 500 (out of 50,000) Iterated Function Systems sampled according to Algorithms 1 and 2, and used to pre-train the models for which we report results in the paper. We show just the binary-rendered fractal images (without color or background) to give a clear picture of the fractal geometry.

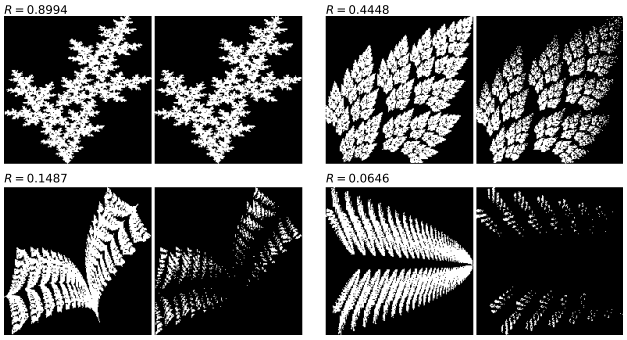

Figure 13. Rendered IFS codes using different probabilities (determinant-based on the left, uniform on the right of each pair). Uniform probabilities don't work well when the determinants of the system have significantly different magnitudes. R is the ratio of the smaller to the larger determinant.

D.4. System Probabilities p_i

An affine IFS code consists of a set of affine functions, each with an associated sampling probability (see Section 3). The sampling probabilities p_i don't affect the shape of the underlying attractor, but they do influence the distribution of points on the attractor that are visited during iteration. Figure 13 shows several IFS rendered using two different choices for p_i: (1) p_i is proportional to the magnitude of the determinant of the linear part of the transform, $p_i \propto |\det A_i|$; and (2) p_i is uniform, $p_i = \frac{1}{|S|}$. When one determinant is significantly larger than the other, there are parts of the attractor that don't get visited during iteration using uniform p_i. We use the determinant method for setting p_i in all our experiments.

E. First Layer Filters

In Figure 10, we show a comparison of the filters from the first layer of ResNet50, pre-trained using different methods. Interestingly, it appears that filters learned from multi-instance prediction are closest to those learned by pre-training on ImageNet.

F. Additional Results

Here we include some additional experimental results that didn't fit in the main body of the paper. Our main set of experiments evaluated fine-tuning performance using image resolution 224×224. One common way to achieve better performance is to use a larger image resolution, such as 448×448. We fine-tuned on CUB using this resolution, and the results are shown in Figure 14. We see better performance across the board, with FractalDB now outperforming training-from-scratch, and with the relative performance order otherwise staying the same. At the higher resolution, we also see the gap between ImageNet and fractal pre-training

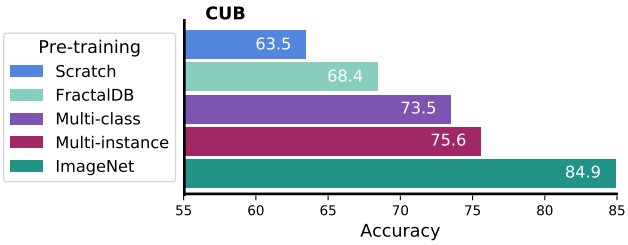

Figure 14. Results of fine-tuning on CUB using a larger image resolution (448×448). The pre-trained networks are the same as in Figure 6.

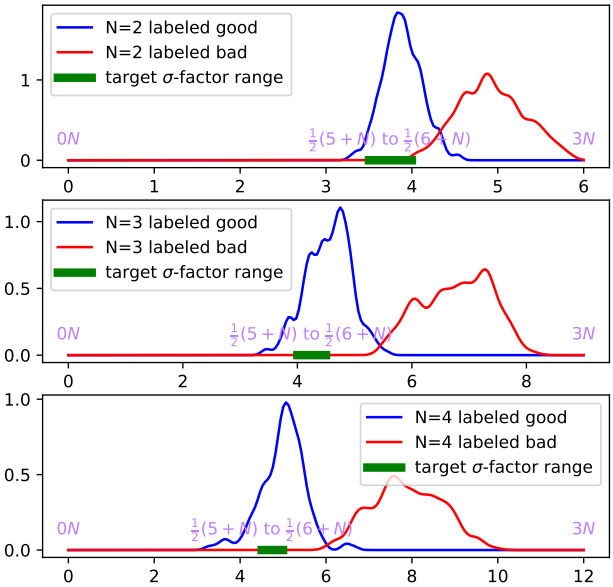

Figure 15. Plot of σ-factor densities of hand-labeled systems with $N = 2, 3, 4$. It is important to note that the $\frac{1}{2}(5+N)$ to $\frac{1}{2}(6+N)$ range was chosen well before these plots were generated; the plots strongly validate the selected range.

get wider, indicating there is still plenty of work to do to improve the fractal pre-training methods.

G. σ-factor Density for Hand-labeled Systems

In Figure 15, we show the distribution of σ-factors for the hand-labeled systems discussed in Section 3.2.2. For each value of $N \in \{2, 3, 4\}$, several hundred systems were labeled as to whether or not they had subjectively "good" geometry. It is critical to point out here that these plots were generated only *after* the range of $\frac{1}{2}(5+N)$ to $\frac{1}{2}(6+N)$ was determined empirically; however, the plots strongly validate the range selected.

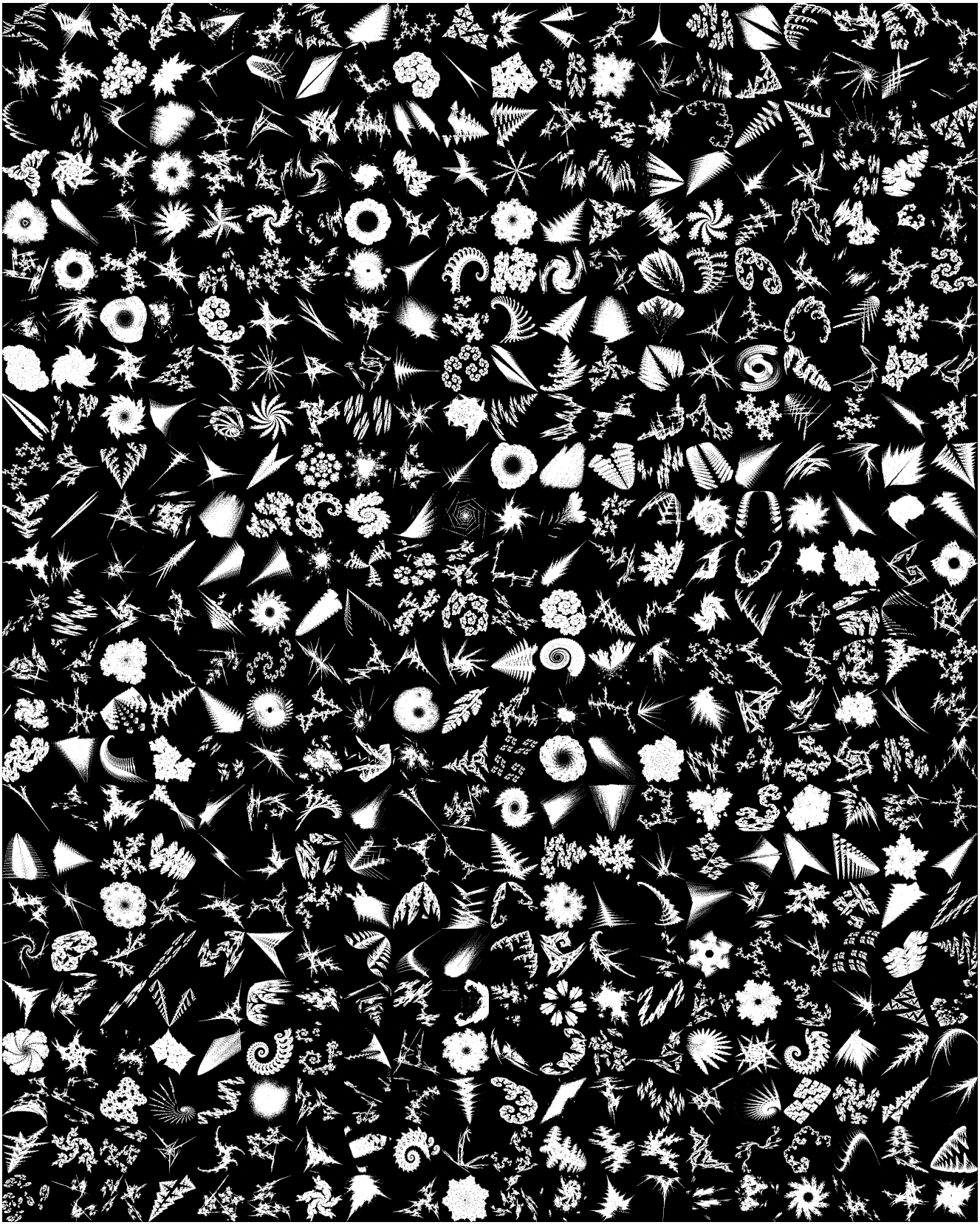

Figure 16. Example images from 500 different systems used in our fractal pre-training.