On Lindenmayer Systems and Autoencoders

Andrej Lucny

Comenius University, Bratislava 84248, Slovakia,
lucny@fmph.uniba.sk,
WWW home page: http://dai.fmph.uniba.sk/w/Andrej_Lucny/en

Abstract: Lindenmayer Systems can serve to deep learning not only as a generator of simulated datasets. They could provide datasets of images generated from a very few parameters that enable us a better study of the latent space, which is crucial for the majority of deep neural networks. We process a dataset, generated by a parametric Lindenmayer system, with the convolutional autoencoder. We aim to recognize the values of the Lindenmayer system parameters by its encoder part. Finally, we partially turn a generator based on its decoder part to a neural network that generates images from the dataset upon the Lindenmayer system parameters.

1 Introduction

Deep neural networks for the vision we typically train from datasets of annotated images. Their preparation is a manual job that sometimes we can avoid if we use the so-called simulated dataset. There are several grammar-based systems that we can use for the simulation [5]. One of them is Lindenmayer systems [8], used already for this purpose [14]. However, this approach issues many other questions that we would like to deal with in this paper. Can we find the parameters of the Lindenmayer system somewhere inside the neural network that is processing a dataset produced by the Lindenmayer system? Can we create a neural network that generates the same images as the Lindenmayer system? And could it make them from the parameters of the Lindenmayer system?

1.1 Parametric Lindenmayer Systems

We employ the parametric Lindenmayer system proposed in [10]. It generates rose leaves upon eight parameters, from which just two can significantly vary: the angle of the stem of the rose leaf and the angle between the left and right venations and the stem. It has a set of production rules which are applied on an initial axiom iteratively and per iteration simultaneously (Table 1). As a result, the system generates strings in Table 2.

We turn the generated strings into images in two steps. At first, we use turtle graphics following symbols G, $+$ and $-$ (go, rotate left, rotate right) structured by $[\,]$. In this way,

Table 1: Production rules for rose-leaf images, borrowed from [10], page 126.

ω_0	:	$[\{A(0,0).\}][\{A(0,1).\}]$
$p1$	:	$A(t,d) : d = 0 \rightarrow .G(LA,RA).$ $[+B(t)G(LC,RC,t).\}][+B(t)\{.]A(t+1,d)$
$p2$	:	$A(t,d) : d = 1 \rightarrow .G(LA,RA).$ $[-B(t)G(LC,RC,t).\}][-B(t)\{.]A(t+1,d)$
$p3$	:	$B(t) : t > 0 \rightarrow G(LB,RB)B(t-1)$
$p4$	:	$G(s,r) \rightarrow G(s*r,r)$
$p5$	:	$G(s,r,t) : t > 1 \rightarrow G(s*r,r,t-1)$

$d = 0$ means the left side and $d = 1$ the right side of the leaf. t is timing. $G(length, growth\_rate)$ corresponds to venations. $+$ and $-$ represent rotation. $[$ and $]$ define a tree structure of the generated string. Dots represent points on the leaf that are structured to polygons by $\{\}$. LA, LB, LC are parameters for the initial length of the main segment, the lateral segment, and the marginal notch. RA, RB, RC represent their growth rate. $+$ and $-$ have also parameter - the angle between stem and venations. The last parameter is the direction of the stem, that we select when we interpret the string and turn it into an image.

Table 2: Strings that represent rose leaves, generated by the Lindenmayer system. From top to bottom: axiom, the first, the second, and the third iteration.

$[\{A(0,0).\}][\{A(0,1).\}]$

$[\{.G(5,1.15).[+B(0)G(3,1.19,0).\}][+B(0)\{.]A(1,0).\}]$
$[\{.G(5,1.15).[-B(0)G(3,1.19,0).\}][-B(0)\{.]A(1,1).\}]$

$[\{.G(5.75,1.15).[+B(0)G(3,1.19,0).\}][+B(0)\{.].$
$G(5,1.15).[+B(1)G(3,1.19,1).\}][+B(1)\{.]A(2,0).\}]$
$[\{.G(5.75,1.15).[-B(0)G(3,1.19,0).\}][-B(0)\{.].$
$G(5,1.15).[-B(1)G(3,1.19,1).\}][-B(1)\{.]A(2,1).\}]$

$[\{.G(6.6125,1.15).[+B(0)G(3,1.19,0).\}][+B(0)\{.].$
$G(5.75,1.15).[+G(1.3,1.25)B(0)G(3,1.19,1).\}]$
$[+G(1.3,1.25)B(0)\{.].G(5,1.15).[+B(2)$
$G(3,1.19,2).\}][+B(2)\{.]A(3,0).\}][\{.$
$G(6.6125,1.15).[-B(0)G(3,1.19,0).\}][-B(0)\{.].$
$G(5.75,1.15).[-G(1.3,1.25)B(0)G(3,1.19,1).\}]$
$[-G(1.3,1.25)B(0)\{.].G(5,1.15).[-B(2)G(3,1.19,2)$
$.\}][-B(2)\{.]A(3,1).\}]$

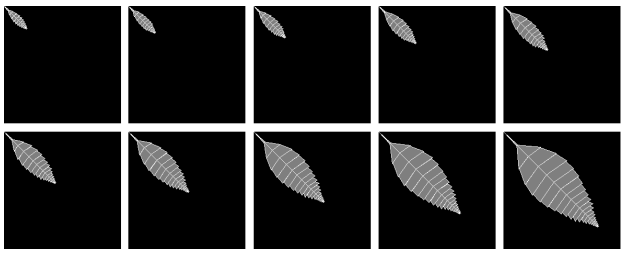

Figure 1: Rose-leaf images generated by the Lindenmayer system.

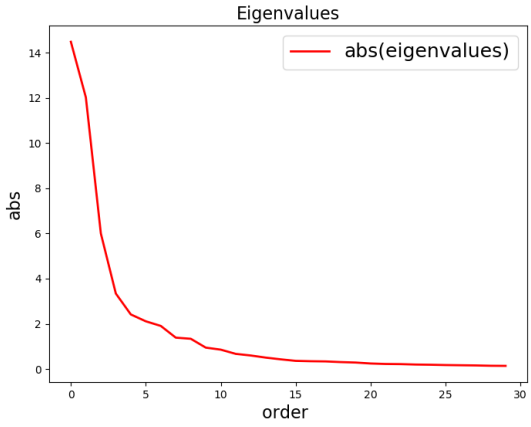

Figure 2: Eigenvalues confirms the fact that the generated dataset has a few amount of parameters.

we calculate the exact positions of all dot symbols that represent points. In the second step, we structure these points into polygons following symbols { } and draw the polygons. As a result, we get an image containing a rose leaf (Figure 1). Varying the parameters of the Lindenmayer system, we can generate the whole dataset of such images.

Having any dataset, we can get imagination about the number of parameters, that generate it, via Principal Component Analysis (PCA) [9]. We concern that two-dimensional images are just one-dimensional vectors, i.e. we put their pixels row by row to one line. Thus we turn the dataset of 28x28 images to a set of 784-dimensional vectors. Then we can calculate their covariation matrix and find its eigenvectors. Following the corresponding eigenvalues, we can find that much less than 784 eigenvectors is significant. In our case, it is enough to concern from 8 to 16 eigenvectors (Figure 2). Now, we can express each image from the dataset as a sum of the mean and multiples of the eigenvectors (Figure 3). We can also make a generator that turns manually selected values of the eigenvector multipliers to images, but its quality regarding the generation of rose leaves is low.

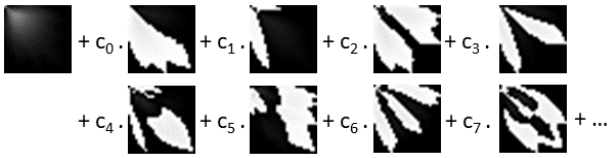

Figure 3: Any image in the dataset can be expressed as a sum of the mean and multiples of the eigenvectors.

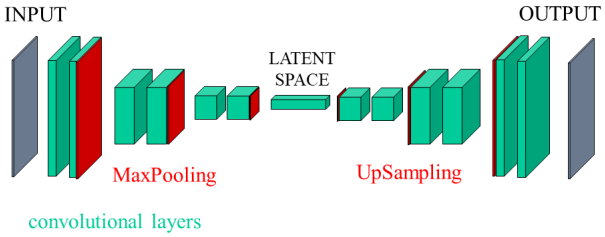

Figure 4: Autoencoder.

1.2 Deep learning and Autoencoders

Deep learning [4] is young but well-known and very successful part of machine learning based on artificial neural networks with a specific architectural design. They enhance the classic neural networks like the perceptron [12] that are theoretically strong, but processing larger inputs as images is not a tractable task form them in practice. Deep neural networks typically employ gradual decreasing of data dimension and turn the input image to a feature vector which has a small enough dimension to be further processed classically. This approach is reflected in their architecture by a deep sequence of convolutional layers (which usually implement 3x3 or 5x5 kernel-based operators) interlaced with the MaxPooling layers (which are responsible for the dimension reduction since they replace e. g. 2x2 values by their maximum) followed by a few fully connected layers corresponding to the classic perceptron. The features do not need to be designed manually but they are found automatically in the process of end-to-end training [13] that corresponds to minimalization of a suitable loss function. The feature vector can be concerned as a point in the so-called latent space. We wish that similar images are mapped to close points and different images to far points in that space. We also want that the feature vector would contain as much information about the corresponding image as possible. The trick on how to push the neural network to learn such feature extraction is the core of the whole deep learning. It can be demonstrated on a neural network called autoencoder (Figure 4).

Autoencoder not just reduces the dimension of the input data into the feature vector but then performs an opposite process and expands the data to their original size, using UpSampling layers (which replace each value with its e.g. 2x2 copies). Then we train it to provide the same output on a given input. If we succeed, then we are sure that the feature vector represents the input image well, because

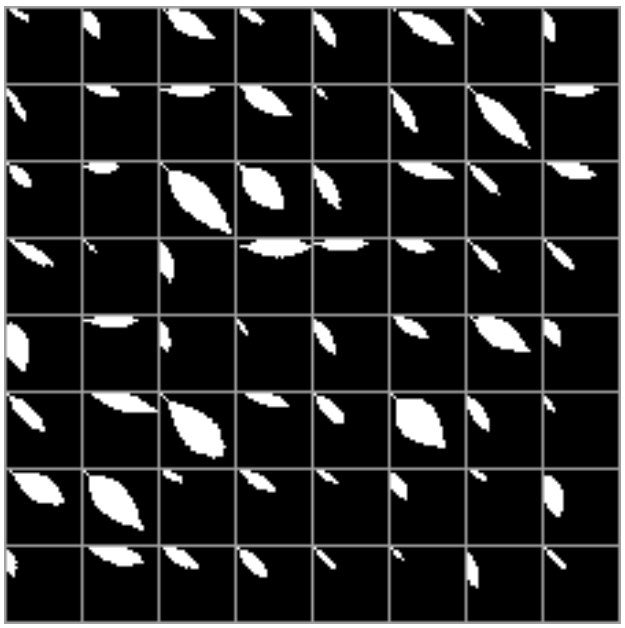

Figure 5: A few samples from the generated dataset. The images are annotated by parameters of the Lindenmayer system used for their generation.

it is possible to generate the image from the feature vector. After such training, we can cut the autoencoder into two parts: encoder and decoder. The encoder turns images to feature vectors and can be combined with a perceptron to provide classification or detection tasks. The decoder turns feature vectors to images and can be used as a generator of images, even such images which have been never presented to the network.

Of course, typically it is difficult to understand representation in the latent space when we are working with real images that have many parameters. Will it be more simple if we present to the autoencoder dataset precisely generated from a very concrete and small number of parameters (what Lindenmayer systems can do for us)? The organization of the latent space is crucial as it is shown by its advanced versions like the variational autoencoder [6]. Therefore we would like to play with this idea a bit. In the next chapters, we prepare a suitable dataset (chapter 2), we train an autoencoder and compare the set of the images with the set of their feature vectors (chapter 3), try to recognize parameters of the Lindenmayer system by the encoder (chapter 4) and turn decoder from a generator based on feature vectors to a generator based on the parameters of the Lindenmayer system (chapter 5).

2 Dataset preparation

We have employed the Lindenmayer system defined in Table 1 for generating our dataset of rose-leaf images. We have implemented the Lindenmayer system in Python 3.6 using OpenCV 4.3.0 [1]. Concerning simplicity, we have been varying mainly the stem angle and the angle between stem and venations. Other parameters can vary just slightly; the resulted image is far from a rose leaf otherwise. We have also turned the output images to binary form and resized them to 28x28. That enables us to use a proven autoencoder architecture, which requires this input size.

We have decided that the stem always starts in the top left corner. This decision enables us to process the dataset also with straightforward methods like eigenimages and compare their results with the autoencoder. All together our dataset had 1498 images. A few samples can be seen in Figure 5. Of course, we have recorded also the parameters which we have used for the generation of each image. In this way, we have created an annotated dataset free of charge.

3 Autoencoder training

Involving deep learning, we start with the training of the autoencoder. Thus, so far, we will not work with the image annotation. We utilize a proven architecture of autoencoder from [3] [11]. On the input, the neural network receives grayscale images (pixels in range 0.0-1.0). They are processed by a block of sixteen convolutional layers with kernels 3x3, then the dimension is reduced by MaxPool layers. The output is processed by the next eight convolutional layers and again reduced. And this repeats until the input data shape 28x28x1 is turned trough 28x28x16, 14x14x8, 7x7x8, 4x4x8 to which is the feature vector. Then like in the mirror, we expand the data by convolutional and UpSampling layers to the original size 28x28x1 (Figure 6).

For non-linearity, the convolutional layers use the ReLU activation function, besides two places. There is sigmoid used just before the latent space to ensure that values in the latent space are from interval <0.0,1.0>. And there is sigmoid on the output from the network; not only to enable us to interpret the output as an image with pixels in range 0.0-1.0 but also to enable us to use the binary cross-entropy loss function, which has a better performance than the classic MSE.

We train the autoencoder with Keras 2.3.1 [3] using Tensorflow 2.1.0 as a backend. We use the Adadelta batch gradient descent algorithm. After 200 epochs, the accuracy is 98.38% on the training set and 98.60% on the testing set (10% of samples) (Figure 7). The achieved quality is good (Figure 8). Now, the autoencoder can code each image to a vector of 128 floats 0.0-1.0 and decode the vector to a very similar image (Figure 9) and we can continue with its splitting into two parts: encoder and decoder.

While we can employ the encoder part for generating another dataset that contains the feature vectors, the decoder part can be used as a generator of rose-leaf images. It is not a very handy generator since we have to set up properly 128 values 0.0-1.0, but it is possible to gener-

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 8)         1160
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 8)           0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 8)           584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 8)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 128)               0
_________________________________________________________________
activation_1 (Activation)    (None, 128)               0
_________________________________________________________________
reshape_1 (Reshape)          (None, 4, 4, 8)           0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 4, 4, 8)           584
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 8)           584
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 14, 14, 16)        1168
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 28, 28, 16)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 1)         145
=================================================================
Total params: 4,385
```

Figure 6: The architecture of the used autoencoder in details [11].

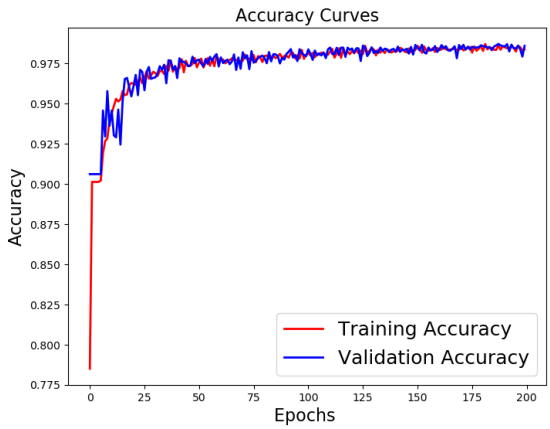

Figure 7: Training of the autoencoder.

ate someting like a leaf just from the feature vector values (Figure 10, on the left).

Can we make such a generator handier? Yes, we can - in a similar way, how we created the generator based on eigenimages. We perform PCA on the dataset of the feature vectors, and we set up just the main components. We express the feature vector as a sum of mean and multiples of eigenvectors. Then we need to set up manually just the multipliers of a few significant eigenvectors. We

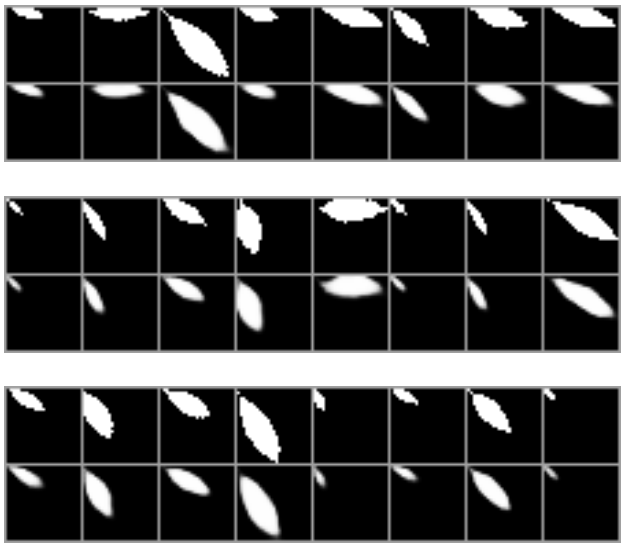

Figure 8: Sample input images from our dataset in the top line and the corresponding output images calculated by our autoencoder.

```
[0.98625815, 0.66299981, 0.99246186, 0.57722825, 0.5       ,
 0.90062493, 0.93622261, 0.5       , 0.5       , 0.95261234,
 0.99352407, 0.5       , 0.5       , 0.98512369, 0.5       ,
 0.82000422, 0.54893059, 0.98727709, 0.71470815, 0.5       ,
 0.5       , 0.94741422, 0.5       , 0.95114863, 0.75802684,
 0.9719044 , 0.5       , 0.5       , 0.5       , 0.82802659,
 0.51641005, 0.97730321, 0.9978047 , 0.80023605, 0.99893409,
 0.6468786 , 0.5       , 0.94131184, 0.99782324, 0.88698453,
 0.5       , 0.5       , 0.99995816, 0.50391984, 0.5       ,
 0.5       , 0.50378138, 0.5       , 0.5       , 0.78034681,
 0.99993455, 0.53947443, 0.5       , 0.5       , 0.5       ,
 0.5       , 0.5       , 0.98526198, 0.91212052, 0.5       ,
 0.5       , 0.87080342, 0.5       , 0.66338164, 0.99515474,
 0.83904874, 0.99057883, 0.5       , 0.5       , 0.95147383,
 0.99873096, 0.96877152, 0.77010125, 0.5       , 0.99994564,
 0.96718907, 0.5       , 0.5       , 0.99889612, 0.5       ,
 0.5       , 0.5       , 0.99997294, 0.88351119, 0.5       ,
 0.5       , 0.86901689, 0.5       , 0.5       , 0.5       ,
 0.9869802 , 0.80338162, 0.5       , 0.5       , 0.5       ,
 0.5       , 0.91442615, 0.8813709 , 0.5       , 0.5       ,
 0.5       , 0.81748968, 0.93058556, 0.98220086, 0.87647492,
 0.5       , 0.97868299, 0.95324636, 0.5       , 0.77138752,
 0.99827719, 0.88651741, 0.5       , 0.5       , 0.99581003,
 0.99394023, 0.5       , 0.5       , 0.99652219, 0.5       ,
 0.5       , 0.5       , 0.90829074, 0.9915418 , 0.5       ,
 0.5       , 0.5       , 0.5       ]
```

Figure 9: An example of an input image from the dataset, its feature vector in the latent space of the autoencoder, and the corresponding output image.

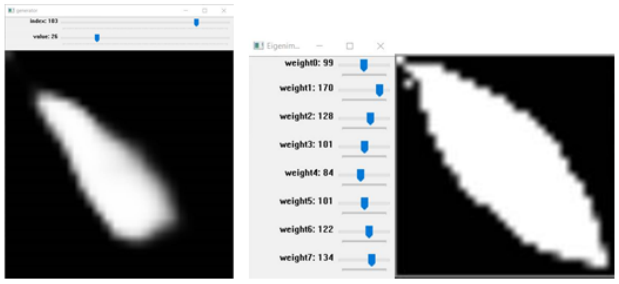

Figure 10: On the left: An image generated from the manually selected 128 values of the feature vector. Quality is quite poor. On the right: An image generated from 8 most significant multipliers of the latent space eigenvectors. Quality is better.

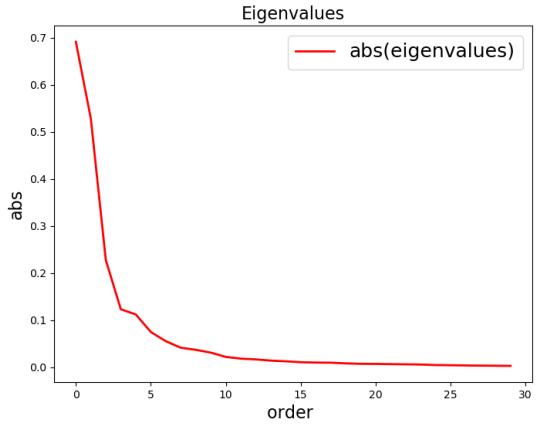

Figure 11: Eigenvalues of the latent space enlighten that encoder does not reduce the number of parameters. (Compare to Figure 2)

have used only eight parameters from which we calculate the 128 items of the feature vector and that we put into the decoder to obtain the corresponding image. This generator is handier, and it provides pretty rose leaves (Figure 10 on the right), though not only them.

4 Recognition of the Lindenmayer system parameters

Though we can generate rose leaves from a few parameters now, it is hopeless to look for the parameters of the Lindenmayer system among them. Neither parameter of the latent space nor multiplier of its eigenvectors directly corresponds to a parameter of the Lindenmayer system. Even when we perform the PCA over the set of the feature vectors calculated by the encoder from images in the dataset, we find that it has the same distribution of the main components (Figure 11).

However, we can easily reveal that they are not so far from them. In the beginning, we aimed to train a perceptron to map the feature vectors to the Lindenmayer system parameters. Though the approach was operational, later we found that it was over-engineered. The linear regression can provide here results as good as the perceptron. In both cases, we can sufficiently recognize the stem angle: 97% by regression and 99% by the perceptron. On the other hand, the angle between stem and venations we have failed to recognize. It is perhaps due to small resolution and binary form of images that is a limitation coming from the used architecture and our hardware.

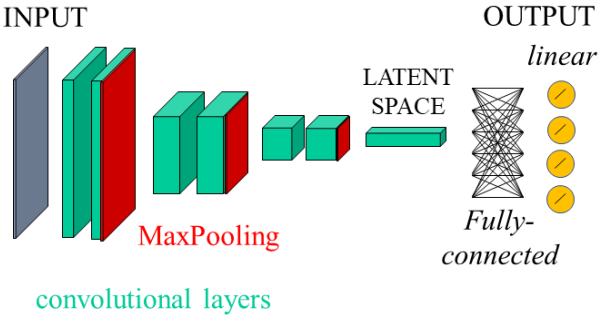

Figure 12: The architecture of the recognizer of the Lindenmayer system parameters.

Linear regression can be added to the encoder neural network as one fully connected layer without bias and with the linear activation function. In this way, we have constructed a neural network that gets an image generated by the Lindenmayer system and recognizes the values of the Lindenmayer system parameters (Figure 12).

5 Neural network generating images from the Lindenmayer system parameters

Though recognition of the Lindenmayer system parameters from the feature vector is straightforward, the inverse operation is not. It is even clear without a trial. However, we can still train a perceptron that approximates the inverse relation. We put all the eight parameters from the annotation of our dataset (two of which significantly vary and six almost constant) to the perceptron input and expect the corresponding feature vector (128 values) calculated by the encoder from the dataset image. Then we search for a suitable number of hidden layers and suitable numbers of neurons in those layers. We have trained each such candidate architecture. We have followed namely validation loss since the accuracy was very low (up to 40%). Fortunately, this does not mean that the trained network does not work, because some items of the feature vector are less important than others, and the error on them can be high without a bad impact. Finally, we have used a perceptron with two hidden layers with the hyperbolic tangent activation function, each containing 256 neurons. And when we joined the perceptron and the decoder, we have got a neural network (Figure 13) that can generate images from

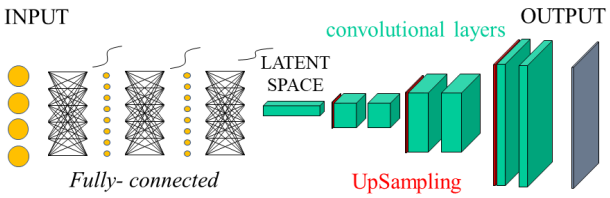

Figure 13: The architecture of the generator of images from the Lindenmayer system parameters.

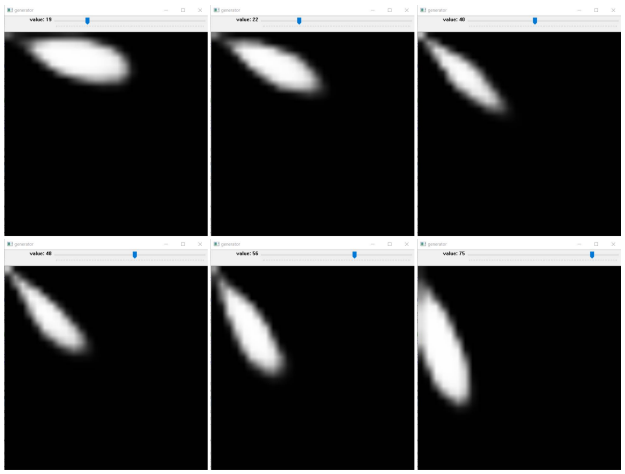

Figure 14: Generating images from the Lindenmayer system parameters.

the parameters of the Lindenmayer system, namely from the stem angle (Figure 14).

6 Conclusion

In this paper, we have dealt with the potential of Lindenmayer systems to pose attractive questions related to deep learning. We have prepared a dataset generated by the Lindenmayer system. Thus we have got its annotation in the form of the Lindenmayer system parameters free of charge. Then we used the dataset for the training of the convolutional autoencoder. Further, we have investigated the relationship between its latent space (feature vectors) and the Lindenmayer system parameters. We found that at least some parameters of the Lindenmayer system we can easily recognize from feature vectors. Finally, we have tried to create a neural-network-based generator analogical to the Lindenmayer system, i.e. a neural network that generates the same images as the Lindenmayer system from the Lindenmayer system parameters. This last job was successful just partially. Our future work should concentrate on the hyper-parameters of the autoencoder architecture. We need an operational architecture that has a larger input image and the latent space as small as possible, containing just parameters that directly correspond to the Lindenmayer system.

All codes developed during the preparation of this paper are available at GitHub: https://github.com/andylucny/On-Lindenmayer-Systems-and-Autoencoders.git

Acknowledgement This research was supported by the project VEGA 1/0796/18.

References

[1] Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools, (2000)

[2] Brownlee, J.: Deep Learning for Computer Vision. edition v 1.4, machinelearningmastery.com, 2019

[3] Chollet, F.: Deep Learning with Python. Manning Publications Co., Greenwich, CT, USA, 2017

[4] Goodfellow, I., Bengio, Y., Courville, A. Deep Learning. MIT Press, 2016

[5] Kelemen, J., Kelemenova, A., Mitrana, V.: Towards Biolinguistics. Grammars 4 (2001), pp. 187–292

[6] Kingma, D., Welling, M.: An Introduction to Variational Autoencoders. Foundations and Trends in Machine Learning: Vol. 12 (2019): No. 4, pp 307-392

[7] Krizhevsky, A., Sutskever, I., Hinton, G.: ImageNet Classification with Deep Convolutional Neural Networks. Advances in neural information processing systems 25(2), (2012)

[8] Lindenmayer, A.: Mathematical models for cellular interaction in development. J. Theoret. Biology 18, (1968), pp. 280–315

[9] Pearson, K.: On Lines and Planes of Closest Fit to Systems of Points in Space. Philosophical Magazine. 2 (11), (1901), pp. 559–572.

[10] Prusinkiewicz, P., Lindenmayer, A., Hanan, J.: Algorithmic beauty of plants. NewYork: Springer-Verlag, 1990.

[11] Rosebrock, A.: Deep Learning for Computer Vision with Python. ImageNet Bundle. 2nd edition. PyImageSearch, 2018

[12] Rosenblatt, F.: The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, (1958), pp. 386–408

[13] Rumelhart, D., Hinton G., Williams, R.: Learning internal representations by error propagation. Parallel Distributed Processing. Vol 1: Foundations. MIT Press, Cambridge, MA, 1986

[14] Ubbens, J., Cieslak, M., Prusinkiewicz, P., Stavness, I.: The use of plant models in deep learning: an application to leaf counting in rosette plants. Plant Methods 14(1), (2018).