

# The Inverse of Fractal Generation - Using a Convolutional Neural Network to Predict the Julia Constant from an Image of a Julia Set

Mantas Majeris  
Joris Peters  
Nikolai Herrmann  
Viktor Veselý

July 4, 2021



$$c = 0.234202 - 0.545120i$$

## Abstract

We consider Julia sets, fractals, which are defined by a complex number known as a Julia constant. A convolutional neural network was trained using Adam stochastic gradient descent to approximate the Julia constant, given a finite resolution image of a Julia set. K-fold cross-validation was used to optimize hyper-parameters. The dataset was generated by us. Our results suggest that a reasonably accurate approximation of a Julia set can be computed given a finite representation of a Julia set. However, several factors such as limited resolution and predefined image boundaries may restrict generalizability.

Code Repository: <https://github.com/jorisptrs/Supervised-Julia>

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Julia Sets . . . . .	3
1.2	Useful properties of Julia Sets . . . . .	4
<b>2</b>	<b>Data</b>	<b>4</b>
2.1	Data generation . . . . .	4
2.2	Pre-processing . . . . .	5
<b>3</b>	<b>Model</b>	<b>5</b>
<b>4</b>	<b>Training</b>	<b>7</b>
4.1	Training . . . . .	7
4.2	Regularization . . . . .	7
4.3	K-fold Cross-validation . . . . .	7
4.4	Final training run . . . . .	8
<b>5</b>	<b>Results</b>	<b>9</b>
5.1	Predicting Values . . . . .	10
5.2	Further informal analysis . . . . .	12
<b>6</b>	<b>Discussion</b>	<b>13</b>
6.1	Analysis of results . . . . .	13
6.2	Improvements . . . . .	13
6.3	Future Research . . . . .	14
	<b>References</b>	<b>14</b>
<b>A</b>	<b>Pseudocode</b>	<b>14</b>
<b>B</b>	<b>Pipeline</b>	<b>15</b>
<b>C</b>	<b>Autoencoder</b>	<b>15</b>

# 1 Introduction

Fractals are some of the most visually appealing mathematical objects out there, but in many cases, they can be generated by a very simple process repeated an infinite number of times. For instance, the Sierpinski Triangle can be obtained by simply starting with a filled triangle, dividing it into four identical smaller triangles, removing the middle one, and recursively applying this process on the three remaining triangles.

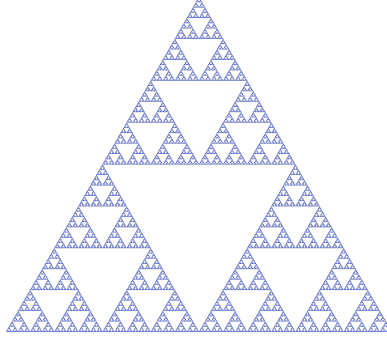


Figure 1: Sierpinski Triangle (Polo, 2007)

Thus, one can obtain a picture of a fractal by repeating a simple process, but this begs the question: can we also guess the process that was used to generate a given picture of a fractal. We will look at a type of fractal, Julia Sets, of which there are infinitely many, and attempt to train a convolutional neural network to approximate its Julia constant (see next subsection for the definition), given only the visual information of the fractal. In other words, we wish to approximate the inverse of generating an image of the Julia set from its constant. Success would indicate that the information behind the generation of Julia sets is present in a finite approximation of the fractal itself.

## 1.1 Julia Sets

A Julia Set is a subset of the complex plane. To determine which numbers on the complex plane belong to a particular Julia Set, we need a few definitions, where  $z, c \in \mathbb{C}$ :

$$f(z) = z^2 + c \quad (1)$$

$$z_{n+1} = f(z_n) \quad (2)$$

We refer to  $c$  in definition 1 as the Julia constant, and each Julia constant will define a distinct Julia Set. To test whether a complex number  $z_0$  belongs to a particular Julia Set, we repeatedly apply the function  $f$  to compute  $z_n$  for ever-larger values of  $n$ . If as  $n$  approaches infinity,  $z_n$  gets further and further away from the origin, then  $z_0$  is not in that particular Julia Set, if  $|z_n|$  stays upper bounded by a finite real number, then  $z_0$  is in that Julia Set.

When generating pictures of Julia Sets in practice, we decide on some finite number of iterations to apply  $f$ , and some large but finite threshold. If the threshold is crossed, we consider the  $z_0$  we picked to not be in the Julia Set. This approach can lead to visually indistinguishable approximations of Julia Sets (see figure 2). Note, that when plotting Julia Sets, we often do not simply have binary values. Instead, if the point did escape the threshold (i.e. the point is not in the set), we change the color based on how many iterations it took to escape. This can be necessary if there are not many points in the Julia Set, and not doing so might make it seem like an empty set.

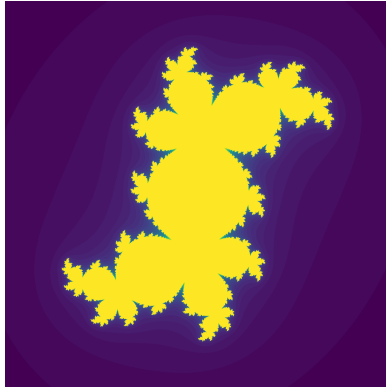


Figure 2: Julia Set with  $c = 0.25 - 0.5i$

## 1.2 Useful properties of Julia Sets

It is helpful to know that Julia Sets are always symmetric about the origin, meaning that if point  $z$  is in a Julia Set, so is  $-z$ . It is very easy to prove that this is the case. Take any point  $z_0$ , and assume that it is in a Julia Set. Note, that  $f(z) = f(-z)$  (see definition 1), meaning after one application of  $f$ , the  $z_n$  will be the same whether we started with  $z_0$  or  $-z_0$ , so if  $z_0$  is in a Julia Set, then  $-z_0$  is also in that same Julia Set.

This will be useful, because it allows us to cut the amount of input data for our neural network in half, without losing any of the information.

Furthermore, Julia Sets are self-similar, which is easy to see upon visual inspection. This is useful to keep in mind because it means we do not need to have an infinite resolution to know what the fractal will look like if we zoom in. This means it should be possible to approximate a higher resolution version of a Julia Set, given a lower resolution image.

## 2 Data

Our dataset consists of a sample  $S = (u_i, y_i)_{i=1, \dots, N}$ , with  $N = 10000$ , of pairs of images  $u_i \in \mathbb{R}^{64 \times 128}$  and Julia constants  $y_i \in \mathbb{R}^2$ , with the two real numbers representing the real and imaginary parts of one complex number. Although the self-similarity of Julia Sets might reduce the needed resolution, it needs to be sufficiently high resolution to capture the patterns that distinguish close fractals from each other. Therefore, we chose a resolution of  $64 \times 128$ , which is relatively high compared to the image sizes typically used in machine learning tasks. Thus, we make use of the symmetry property of Julia Sets to train a network on the information of squared  $128 \times 128$  images using only half the picture.

### 2.1 Data generation

To have the data in a suitable format for our network, and because it is easy to generate random complex numbers, we generate the data ourselves. We use a C++ program to generate the data. First, we sample the complex plane for Julia constants  $y_i$ , then generate a picture approximating a Julia Set  $u_i$  using the process described in the introduction section. We want our sampler to choose points on the complex plane which are no further away from the origin than 2. This is because when we pick Julia constants far away from the origin, we get very sparse and not very detailed Julia Sets. This means these fractals would often look identical using our representation of the fractals.

We use uniform sampling to generate labels for our training data. First, we tried a naive approach by generating a number  $(r, \theta) \in \mathbb{R}^2$ , where  $r \in [0, 2]$ ,  $\theta \in [0, 2\pi]$ .  $r$  represents the distance from the origin, and  $\theta$  represents the angle between the line containing the point and the origin, and the positive real axis. We then intended to convert this point to Cartesian coordinates. However, this approach led to more of a normal distribution around the origin rather than a uniform distribution. This meant our algorithm would have likely been over-trained on points around the origin, and under-trained for

points that were further away. Therefore, we used a variation of a sampling algorithm (sigfpe, 2011). The pseudocode can be found in A of the appendix.

This algorithm leads to a much more uniform distribution:

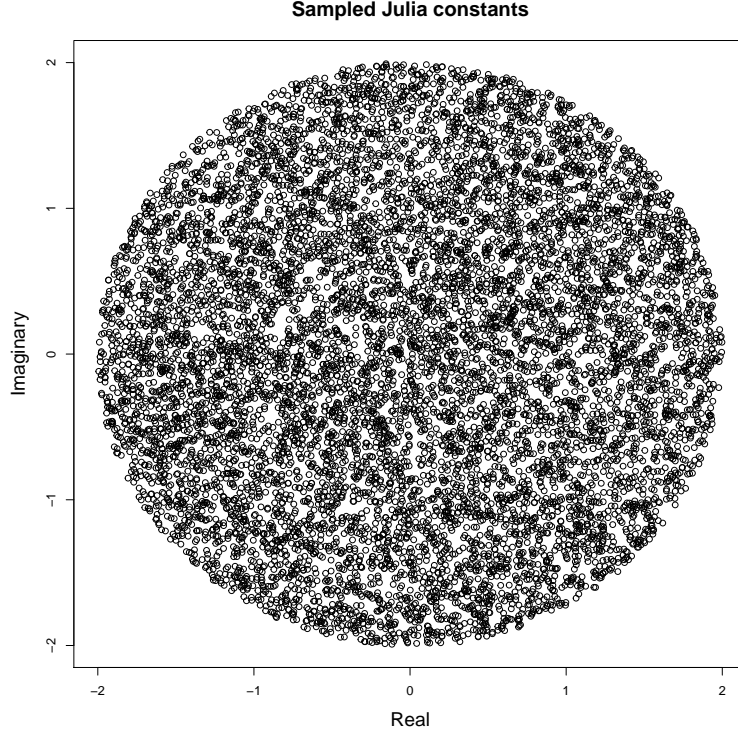


Figure 3: 2000 generated Julia constants with the random sampler

For each generated Julia constant, we generate a picture of the Julia Set which corresponds to that constant. We set the boundaries of the picture to be  $-1.5$  and  $1.5$  for the real part, and  $0$  and  $1.5$  for the imaginary part (again, we could take only half of the imaginary range due to symmetry). We use these boundaries because the real and imaginary parts of most of the points of the fractals seemed to be confined within the  $[-1.5, 1.5]$  range, and expanding the boundaries would result in large 'uninteresting' areas outside of the fractal. We make sure the points we plot are uniformly distributed over the range of the picture. For each point, we test whether it is in the Julia Set using the procedure described in the previous section. We set the escape threshold to  $3$ , and the number of iterations to  $256$ . The value of a pixel in our input vector will be the number of iterations it took for the point represented by that pixel to escape the threshold. If the point did not escape, the value will be  $256$ .

## 2.2 Pre-processing

As a pre-processing step, the values of each feature (pixel) were scaled to a range of  $[0, 1]$  relative to the whole dataset to speed up learning. This min-max scaling was preferred over standardization (to  $\mu=0$  and  $\text{sd}=1$ ) since the features do not follow a Normal distribution.

We do not do any other pre-processing, as we generate our own dataset, and we can tailor the generated data to be ready for learning. We did not normalize the data during the data generation, since stored integers occupy less space than floats. This means we get faster loading times when loading the data as integers. Therefore, it makes sense to store the data without normalizing, and only normalizing after loading.

## 3 Model

We are facing the supervised regression task of predicting two labels: the real and imaginary parts of the Julia constant  $y_i$ , given a  $64 \times 128$  image  $u_i$ . Formally, it amounts to finding a function  $h_{\text{opt}}$

from the function space  $H = \{h \mid h : \mathbb{R}^{64 \times 128} \rightarrow \mathbb{R}^2\}$  such that, given a loss function  $L$ , and a training set of size  $N$ :

$$h_{opt} = \arg \min_{h \in H} \sum_{i=1}^N L(h(u_i), y_i) \quad (3)$$

Given such a high-dimensional input, using a multi-layer perceptron (MLP) would have required a very large number of neurons for good results to be expected. With two fully connected 512-neuron layers, for example, an MLP would already have over 4 million trainable parameters (compared to the 4300 of our eventual architecture). This would be problematic, as not only would training such a network require a very large amount of computational power, but also, the risk of overfitting would increase significantly (O’Shea & Nash, 2015). Therefore, we opted for a convolutional neural network (CNN) which has been successfully applied to many image regression tasks. CNN’s also have the advantage that their convolutional layers are applicable on 2-dimensional (image) data, meaning that the spatial information of our fractals can be retained until the very end of the network, where a linear transformation is applied.

We use a CNN architecture with two convolutional layers, each followed by a max-pooling layer, and, finally, a fully connected linear layer as shown in 4.

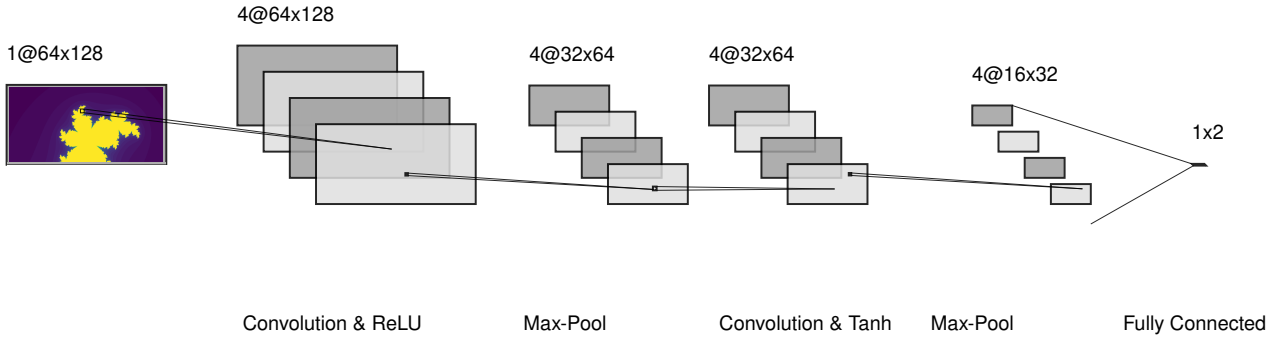


Figure 4: CNN architecture

Firstly, we settled for this relatively simple five-layer architecture rather than some more complex architectures like LeNet5 or AlexNet, which initially seemed like adequate inspirations for this task, but exceed our understanding and would have been computationally too expensive.

The two convolutional layers were each given a kernel size of  $3 \times 3$ , a stride and a padding of 1, and 4 channels. Furthermore, to introduce non-linearity, the first convolutional layer is fed through a rectified linear unit (ReLU; a classical choice in CNNs, that provides a cheap and quick training with a dampened vanishing gradient), and the second is fed through a tanh which maps the outputs of the second convolutional layer to a range of  $[-1,1]$  and lead to a better performance than when using ReLU in that layer. Max pooling was added to reduce the dimensionality and the size of the fully connected layer that linearly combines the signal to the two outputs. Lastly, batch normalization was applied after each convolution to speed up gradient descent.

## 4 Training

To implement our neural network, we used the python PyTorch module (version 1.9.0). A complete list of the dependencies can be found on the above-mentioned repository on Github, while the full training pipeline of our program (Figure 11) can be found in the Appendix below.

### 4.1 Training

To minimize the mean squared distance between the predicted and true Julia constants, the network was trained using the  $L_2$  loss function  $L : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}$ :

$$L(h(u_i), y_i) = \|h(u_i) - y_i\|^2 \quad (4)$$

To iteratively approach the optimal model  $h_{opt}$ , we used Adam, a variant of stochastic gradient descent with an adaptive learning rate. We chose Adam for two reasons. First, to reduce the computational cost on a large dataset compared to normal GD. We do this by approximating the estimated loss with one batch of training examples (with a batch size of 128) at each step of the gradient descent. Second, because adapting the learning rate initially speeds up training while exploring local optima ever so precisely. Although varying throughout training, a base learning rate was picked using k-fold cross-validation.

### 4.2 Regularization

To prevent overfitting, we used weight decay from PyTorch’s Adam implementation. This adaptation penalizes the sum of all model parameters (weights and biases) squared (“L2 Penalty”) (*Adam - PyTorch 1.9.0 Documentation*, 2019). The strength of the regularization is set by the weight decay parameter, whose optimal value was estimated through K-fold cross-validation.

### 4.3 K-fold Cross-validation

To fine tune the hyper-parameters, we used k-fold cross-validation. We examined the base learning rate  $\mu$  ( $\mu \in M = \{0.0001, 0.001, 0.01, 0.1\}$ ) and the weight decay parameter  $\lambda$  ( $\lambda \in \Lambda = \{0, 0.0001, 0.001, 0.01, 0.1\}$ ) in a grid search over the possible combinations of these sets of candidate parameter values  $P = M \times \Lambda$ . The total number of combinations  $|P|$  in our case is 20. For each parameter combination  $p \in P$  we perform a k-fold cross-validation as follows.

Firstly, the dataset  $S$  is split into  $k$  subsets  $S_j$  ( $j = 1, \dots, k$ ). Setting  $k$  to 5 gives us equaled-sized subsets of 400 observations. This procedure was done using the `KFold` class from the `sklearn.model_selection` module. For each fold, one subset is reserved as a validation set ( $V_j$ ) while the others are joined together resulting in the training set ( $T_j$ ). Then, a model is computed using the above procedure on  $T_j$  and validated on  $V_j$  to find the validation risk (Jaeger, 2021). For each combination, the average validation risk can be calculated. Due to rather large amounts of computation 25 epochs were chosen to train the model.

The findings for the hyperparameter optimization are shown in Figure 5; with a smaller learning rate, the validation risk decreases. There is not a clear pattern for flexibility (weight decay).

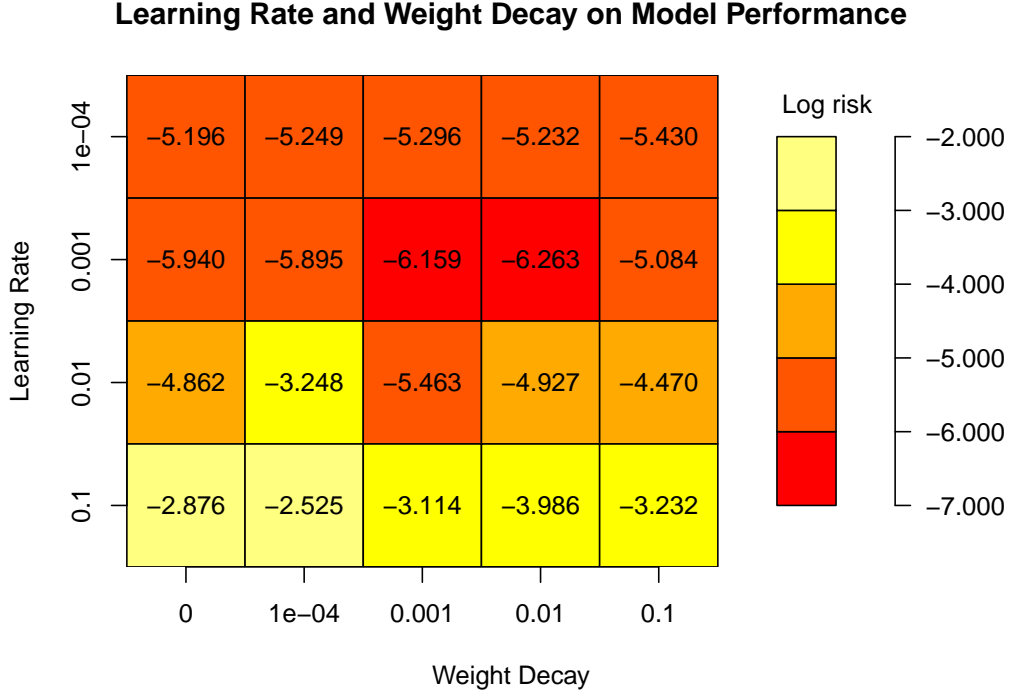


Figure 5: Heat-map matrix showing the average validation risk for each combination of learning rate and weight decay.

The optimal combination  $p$  is determined by the minimum average validation risk. We can see that a model with a learning rate of 0.001 and a weight decay of 0.01 yields the lowest validation risk. We can then proceed to train our final model.

#### 4.4 Final training run

The model was trained on  $N_{train} = 2000$  images, validated on  $N_{val} = 8000$ , and further informal analyses were made on 450 images. Two metrics were saved during training:  $\mathcal{L}_{train}$  (mean training loss) and  $\mathcal{L}_{val}$  (mean validation loss).

$$\mathcal{L}_{train} = \frac{1}{2000} \sum_{i=1}^{2000} L(h(u_i), y_i) \quad (5)$$

$$\mathcal{L}_{val} = \frac{1}{8000} \sum_{i=2001}^{10000} L(h(u_i), y_i) \quad (6)$$

The unusual training to validation set proportion was used to ensure the uniqueness of the validation data. Our data generation algorithm samples Julia constants which are often very close to each other (see Figure 3 for visual explanation). Furthermore, increasing the training dataset size seemed to not affect the model performance as shown in Figure 6; three model runs with different training dataset sizes and all models converged to about the same loss after 20 epochs.



Dataset Size (N) on Model Performance

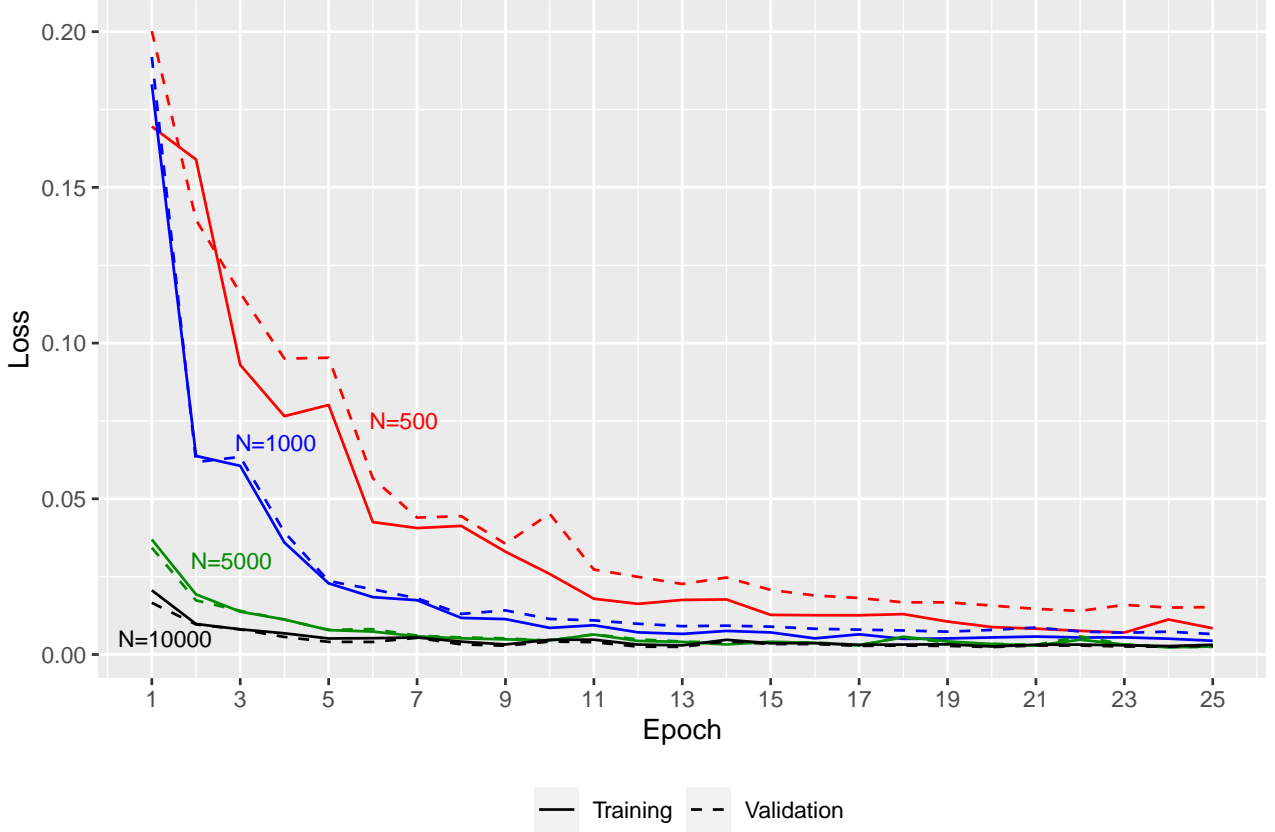


Figure 6: Comparing the MSE Loss between dataset sizes 500, 1000, 5000 and 10000. For each size 80% of the data is used for Training.

The 800 epochs used in the final training are different from the 25 epochs used for cross-validation. This implies that the hyper-parameters found during cross-validation may not be optimal for the final training run. In an ideal world, we would have also used 800 epochs during cross-validation, but this would have exceeded our capacities. However, we assume the parameters found after 25 epochs to be a reasonable approximation.

## 5 Results

Figure 7 shows the plotted results of  $\mathcal{L}_{\text{train}}$  and  $\mathcal{L}_{\text{val}}$ . The network performance stabilizes after approximately 80 epochs on  $\mathcal{L}_{\text{train}} \approx \mathcal{L}_{\text{val}} < 0.0015$ . In the final epoch,  $\mathcal{L}_{\text{train}} \approx 0.000844$  and  $\mathcal{L}_{\text{val}} \approx 0.001002$ . Although this is not the mean euclidian distance between the predictions and the actual Julia constants, these small differences indicate that the predictions come rather close to the true values in the complex plane.

An interesting remark can be made by looking at the  $\mathcal{L}_{\text{train}}$  and  $\mathcal{L}_{\text{val}}$  in Figure 7. The model does not seem to be overfitting even after 800 epochs, since  $\mathcal{L}_{\text{train}} \approx \mathcal{L}_{\text{val}}$ , which suggests that the model actually learns to predict the Julia constants.

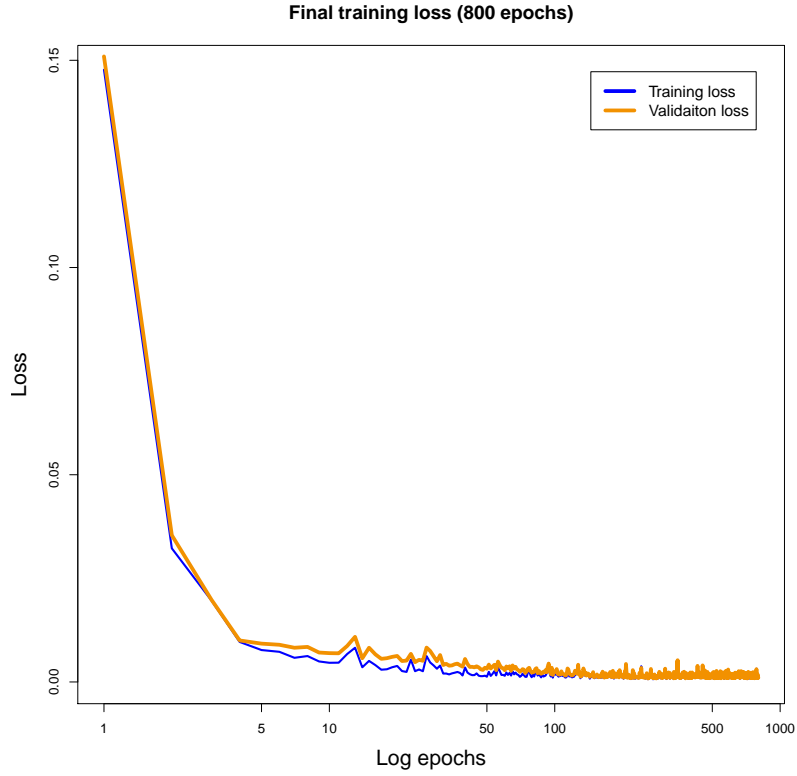


Figure 7: Training and validation MSE loss evaluated on the final model architecture for 800 epochs.

## 5.1 Predicting Values

We shall also inspect the performance of our model visually. Figure 8 compares the plots of three randomly picked data points and the reconstructed (regenerated) plots from the predictions. The reconstructed Julia sets seem to have very similar shapes, but occasionally, as seen from the third example, the reconstruction might be missing some "inside details" of the Julia sets. The last row also has the worst prediction with the predicted constant being 0.0469 units away from the actual constant.

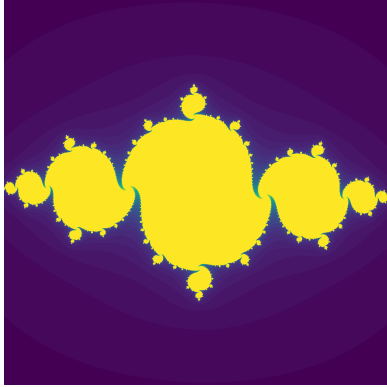
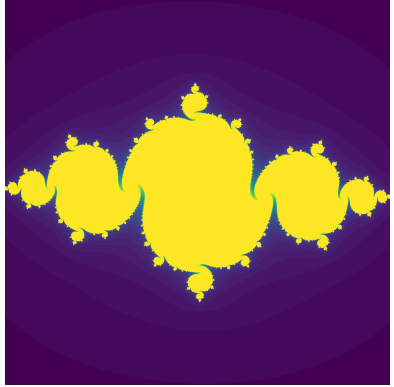
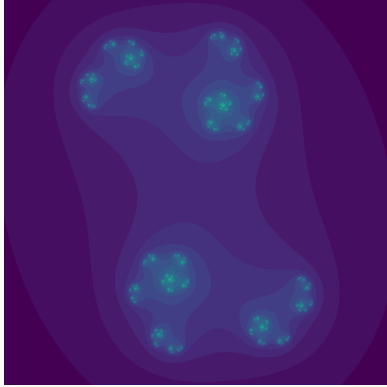
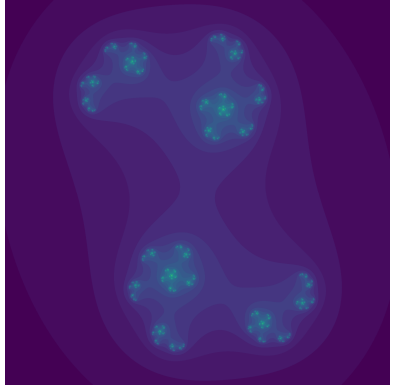
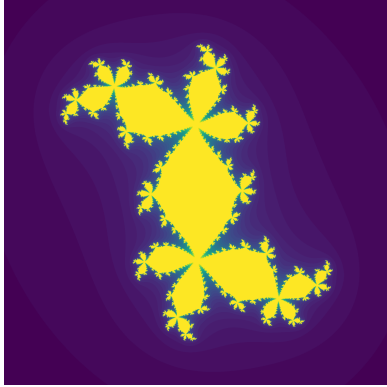
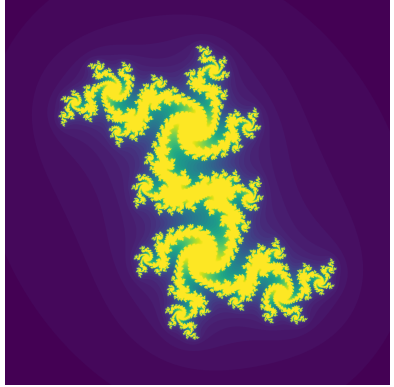
Constant $y_i$	Error $ h(u_i) - y_i $	Input $f(y_i)$	Reconstructed $f(h(u_i))$
$-0.8111 - 0.0647i$	0.0262		
$0.6236 - 0.3956i$	0.0443		
$0.2788 - 0.5307i$	0.0469		

Figure 8: Comparison of the input Julia set and reconstructed Julia set from the predicted constant. The left-most column is the true Julia constant. The Error column is the euclidean distance between  $y_i$  and  $h(y_i)$  (L1 loss for a more intuitive value). The Input column is a higher resolution non-halved version of the input image for the model and the last column is the reconstructed Julia set with the predicted Julia constant, with  $f : \mathbb{R}^2 \rightarrow \mathbb{R}^{2000 \times 2000}$  being a function which plots a higher resolution image given a Julia constant

To inspect the accuracy over the whole sample, Figure 9 shows a comparative plot of the predicted and actual Julia constants (real and imaginary parts). One can observe that our model fitted the data almost perfectly around the origin. The predictions deviate from the straight line (i.e., are less accurate) as the Julia constant is further away from the origin. Additionally, the predictions overall seem to be slightly better for the imaginary component.

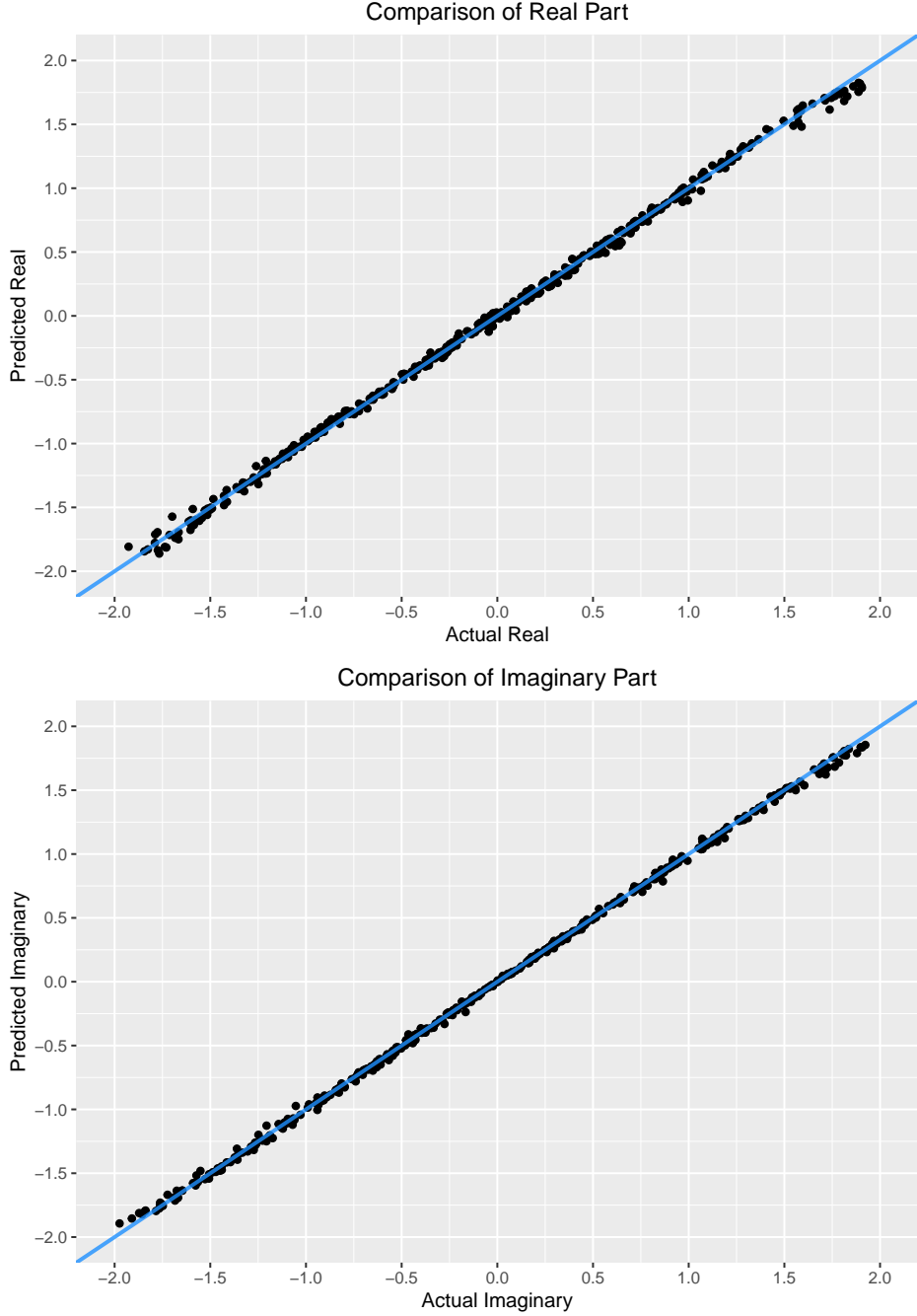


Figure 9: Comparison of predictions and the actual labels using the validation dataset ( $N_{val} = 8000$ ).

## 5.2 Further informal analysis

As mentioned in the Data Generation section, we only consider Julia constants within a radius of 2 from the origin. However, it may be interesting to also perform an informal test to see how the network performs on values outside that radius. We use a deterministic evenly spaced sampler to generate a new smaller testing dataset containing points within a radius of 3 from the origin.

We calculate the loss for each point in the new testing dataset using the previously trained model. We use a linear interpolation to create a surface-colored graph (see Figure 10). The color signifies the loss. Visibly, it drops drastically for Julia constants further away from the origin than 2, which is to be expected, as the network was not trained with values from outside that radius. Also, performance decreases for Julia constants that are further away from the origin (but within a radius of 2 from the origin), possibly, since these will often have important details that are outside of the boundaries we use for our input data.

Furthermore, a few points very close to the origin had a relatively poor prediction. A potential explanation is that Julia constants very close to the origin are less distinct than those that are a little further away.

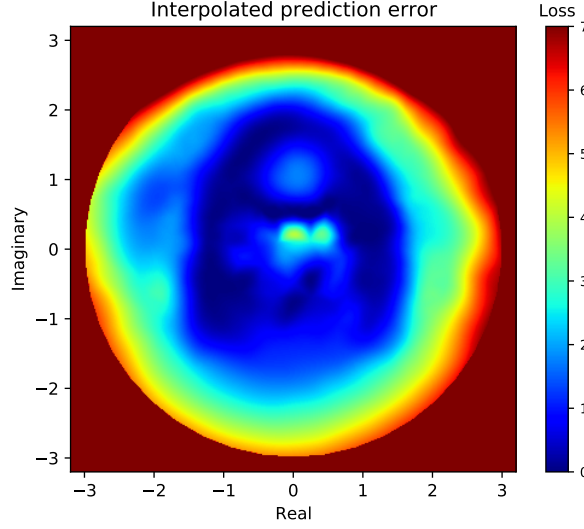


Figure 10: Cubically interpolated loss over the testing dataset. The color signifies the loss at a given point, red being higher loss and blue being lower. The color of points further than 3 away from the origin is a byproduct of the interpolation and does not reflect true loss values.

## 6 Discussion

### 6.1 Analysis of results

The validation loss close to zero suggests, that the neural network was able to closely approximate the Julia constants given the limited resolution pictures. This suggests that the visual information of a Julia Set does, in fact, contain information about the Julia constant, although it is unclear whether this would hold as we increase resolution, and it certainly does not mean that an analytical method of finding a Julia constant given a picture of a Julia Set exists. It is also important to note that it is not always the case that the predicted Julia constant will produce a nearly indistinguishable fractal as the true Julia constant (See Figure 8).

The final mean losses  $\mathcal{L}_{\text{train}}$  and  $\mathcal{L}_{\text{val}}$  are very close, as supported by Figure 7. This suggests that little over-fitting occurred. On the one hand, this may mean that the algorithm successfully learned to recognize patterns within the images. On the other hand, a possible reason for the similarity of  $\mathcal{L}_{\text{train}}$  and  $\mathcal{L}_{\text{val}}$  is that we sampled the Julia constants very densely from the region of radius 2. Additionally, due to the finite resolution of the input images, two Julia constants that are very close to each other may correspond to largely similar pictures, meaning that the training and validation sets may strongly resemble each other.

### 6.2 Improvements

As discussed in our results analysis, the limited resolution may imply a limited applicability of our research. Therefore, redoing the study with more resolution may be worthwhile. If a study with greater resolution pictures found similar results, this would confirm that the visual information within a Julia set can be used to calculate the Julia constant. However, a greater resolution would severely increase the need for computational power. One way to deal with high-dimensional inputs would be to use an autoencoder; due to the self-similarity of the fractals, it might be that very high-dimensional pictures of Julia Sets could be encoded onto lower-dimensional vectors which could then be used as

input for the regression network. The GitHub repository contains one such attempt of ours to apply an autoencoder to reduce the Julia set images of our dataset to a 2-dimensional latent space (some results can be seen in Figure 12 of the Appendix).

### 6.3 Future Research

Our study suggests that images of Julia Sets contain information that was used to generate them. It would perhaps be interesting to see if the same holds for other types of fractals, such as Newtonian Fractals, which are generated by plotting the basins of attraction on the complex plane for different polynomials when applying Newton’s Method (Blanchard et al., 1994).

## References

- Adam - *PyTorch 1.9.0 Documentation*. (2019). Retrieved from <https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam>
- Blanchard, P., et al. (1994). The Dynamics of Newton’s Method. In *Proceedings of Symposia in Applied Mathematics* (Vol. 49, pp. 139–154).
- Jaeger, H. (2021). *Neural Networks (AI) (WBAI028-05)*. Retrieved from [https://www.ai.rug.nl/minds/uploads/LN\\_NN\\_RUG.pdf](https://www.ai.rug.nl/minds/uploads/LN_NN_RUG.pdf)
- O’Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks. *CoRR*, *abs/1511.08458*. Retrieved from <http://arxiv.org/abs/1511.08458>
- Polo, M. (2007). *File:sierpinski triangle.svg*. Retrieved from [https://commons.wikimedia.org/wiki/File:Sierpinski\\_triangle.svg](https://commons.wikimedia.org/wiki/File:Sierpinski_triangle.svg)
- sigfpe. (2011). *Generate a random point within a circle (uniformly)*. Retrieved from <https://stackoverflow.com/questions/5837572/generate-a-random-point-within-a-circle-uniformly>

## Appendix

### A Pseudocode

---

**Algorithm 1** Sampler (sigfpe, 2011)

---

```

1: function NEXTSAMPLE
2:    $\alpha \leftarrow \text{randNum}(0, 2\pi)$ 
3:    $r \leftarrow \text{randNum}(0, 2) + \text{randNum}(0, 2)$ 
4:   if  $r > 2$  then
5:      $r \leftarrow (4 - r)$ 
6:   return  $(r \cdot \cos(\alpha), r \cdot \sin(\alpha))$ 

```

---

## B Pipeline

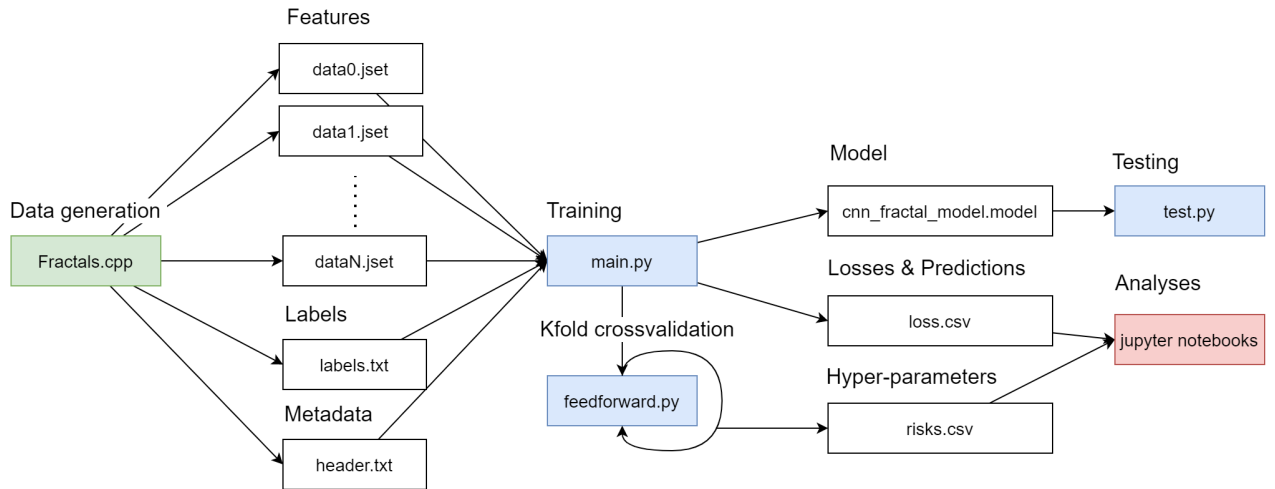


Figure 11: Abstraction of our training pipeline. Green nodes signify c++ language, blue python, red R and whites are plain data files.

## C Autoencoder

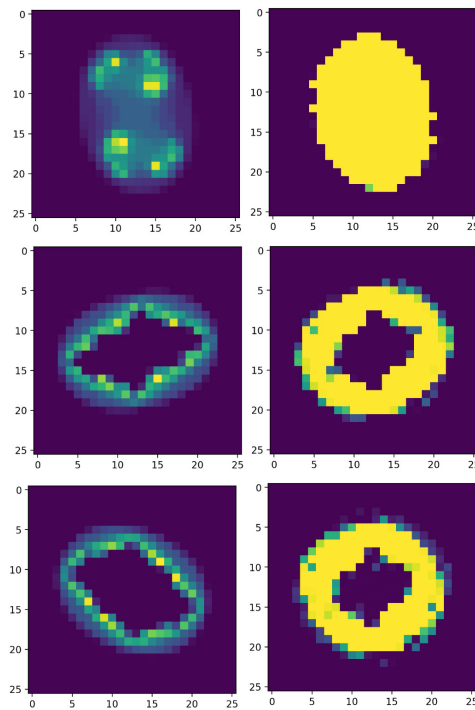


Figure 12: Three example  $28 \times 28$  input images (left) and the corresponding decoded outputs (right) of an autoencoder with a 2-dimensional latent vector and two 512-dimensional linear hidden layers in each its de- and encoder