Real numbers, data science and chaos:
How to fit any dataset with a single parameter

Laurent Boué

SAP Labs

Abstract

We show how any dataset of any modality (time-series, images, sound...) can be approximated by a well-behaved (continuous, differentiable...) scalar function with a single real-valued parameter. Building upon elementary concepts from chaos theory, we adopt a pedagogical approach demonstrating how to adjust this parameter in order to achieve arbitrary precision fit to all samples of the data. Targeting an audience of data scientists with a taste for the curious and unusual, the results presented here expand on previous similar observations [1] regarding expressiveness power and generalization of machine learning models.

Keywords: Chaotic systems ● Machine Learning ● Generalization

Real world data comes in huge variety of shapes and sizes with modalities ranging from traditional structured database schemas to unstructured media sources such as video feeds and audio recordings. Nevertheless, any dataset can ultimately be thought of as a list of numerical values $\mathcal{X} = [x_0, \cdots, x_n]$ describing the data content regardless of the underlying modality. The purpose of this paper is to show that all the samples of any arbitrary dataset $\mathcal{X}$ can be reproduced by a simple differentiable equation:

$$f_\alpha(x) = \sin^2 \left(2^{x\tau} \arcsin \sqrt{\alpha} \right) \tag{1}$$

where $\alpha \in \mathbb{R}$ is a real-valued parameter to be learned from the data and $x \in [0, \cdots, n]$ takes integer values. ($\tau \in \mathbb{N}$ is a constant which effectively controls the desired level of accuracy). Before delving into the logic of how and why f_α is able to achieve such a lofty goal, let us start by a few practical demonstrations. Keeping with the tradition of "fitting an elephant" [2], we start by showing how different animal shapes may be generated by choosing appropriate values of α as displayed in Figure 1.

Figure 1: Animal shapes obtained with the different values of α defined on top of each image. One should consider the data as a scatter plot of pairs of values (x, y) where each $x \in \mathbb{N}$ is associated with a corresponding y value given by $y \equiv f_\alpha(x)$. One goal of the paper will be to show how to find the precise value of $\alpha \in \mathbb{R}$ required to fit any target dataset.
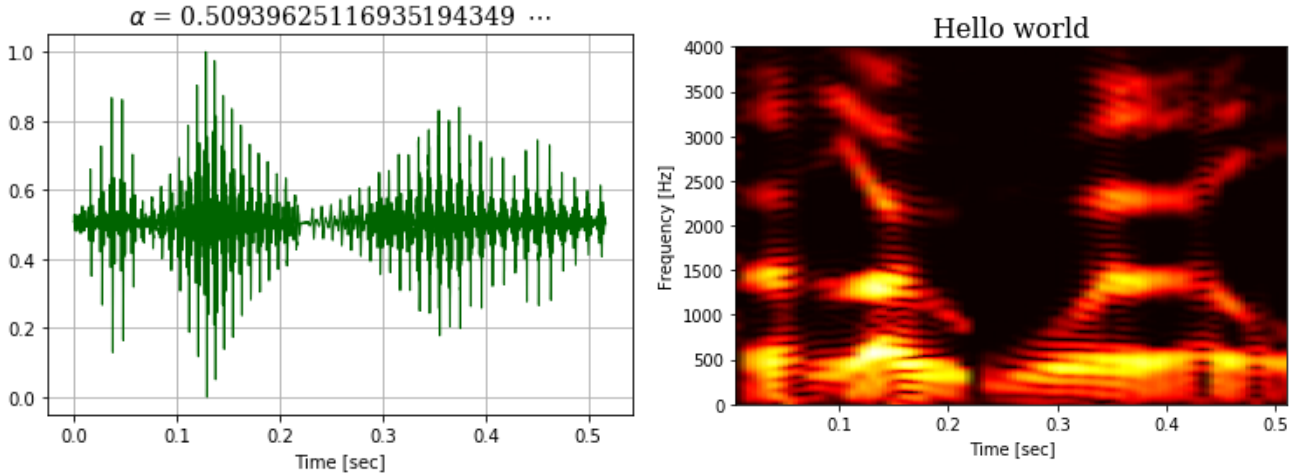
Figure 2: **Left)** Time series obtained by applying f_α at equally-spaced arguments in $\mathbb{N}$. Scaling the horizontal axis by an appropriate sampling rate (≈ 11000 Hz) allows the signal to be interpreted as a sound wave. **Right)** Playing the corresponding audio file reveals a male voice saying "Hello world" characterized by this spectrogram.

Following this demonstration that f_α can generate any kind of doodle-like drawing, let us continue with a literal "Hello world" example in order to further illustrate the capabilities of the approach. Namely, we show in Figure 2 how a well-chosen value of α may be used to produce a complex high-dimensional acoustic signal encoding the actual expression "Hello world"!

Moving on to the connection with data science, let us consider the data modality that has emerged as the epicenter of the current explosion of interest in deep learning: images. Along with the advent of specialized hardware and smarter neural network architectures, it is widely acknowledged that the availability of very large labeled training data has been one of the most important factor responsible for the perceived computer vision "coming-of-age" as manifested by a myriad of academic breakthroughs and integration into successful commercial applications. In this context, the CIFAR-10 dataset continues to stand as one influential yardstick measuring the performance of new learning algorithms. Because of the small resolution of the images, this dataset also serves as a popular research playground. Accordingly, we demonstrate in Figure 3 that it is always possible to find values of α such that f_α builds up artificial images that mirror the categories of CIFAR-10.

Model complexity and generalization? A fundamental question in machine learning research has to do with the evaluation of "model capacity" and its connection to the presumed existence of generalization (even in strongly overparametrized models). Traditional approaches tend to fall in categories such as VC dimension and Rademacher complexity... Nevertheless, in the absence of a generic theoretical framework valid for modern deep learning architectures, it is not uncommon for practitioners to simply count the number of parameters and treat it as a proxy for expressiveness power; an approach somehow inspired by classical information theory based estimators such as AIC and BIC.

The examples above have demonstrated that an elegant model f_α with a simple and differentiable formulation (composition of a few trigonometric and exponential functions) is able to produce any kind of semantically-relevant scatter plot, audio or visual data (text may also be constructed using an identical approach) at the cost of a single real-valued parameter. Without yet revealing all the tricks, one aspect should already be obvious by now: all the information is directly encoded, without any compression or "learning", into $\alpha \in \mathbb{R}$. As mathematical objects, real numbers are non-terminating and therefore contain an infinite amount of information (in particular, they should not be confused with whatever finite-precision data types programming languages may implement) [3]. As such, no generalization can be expected from f_α and indeed we conclude the paper by showing this explicitly in the context of time-series in Figure 9.

In addition to casting doubts on the validity of parameter-counting methods and highlighting the importance of complexity bounds based on Occam's razor such as minimum description length (that trade off goodness-of-fit with expressive power), we hope that f_α may also serve as **entertainment for curious data scientists** [4].
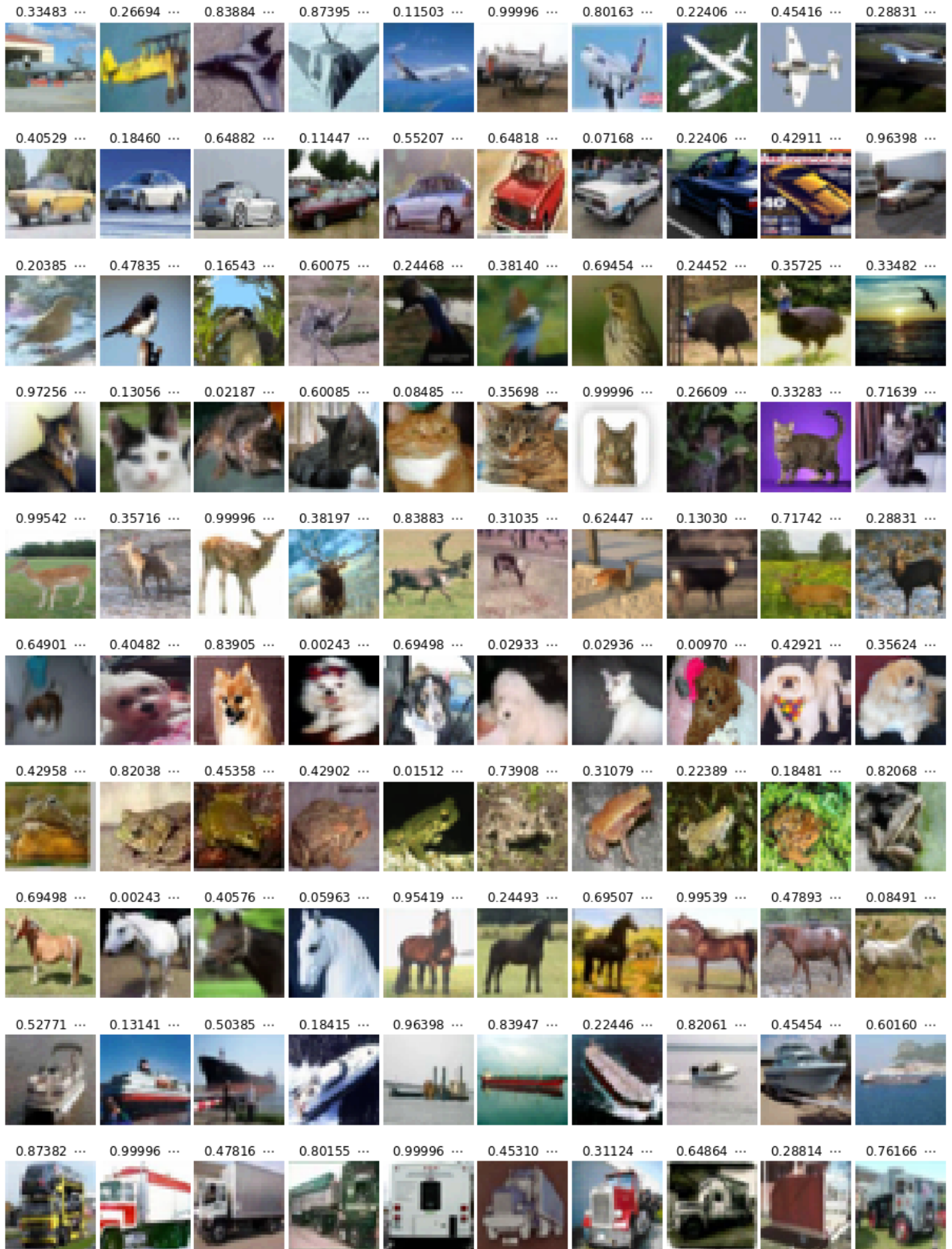
2

Figure 3: Small resolution images are generated by applying f_α for 3072 integer-spaced arguments in $\mathbb{N}$. Folding the resulting list of numerical values appropriately into 3d arrays of shapes $32 \times 32 \times 3$ allows one to produce color images that look like they were drawn from the CIFAR-10 classes: {*airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck*}.

3

Setting up a few essential building blocks. After revealing that α directly encodes the entire dataset $\mathcal{X}$ into a single parameter, the rest of the paper will be dedicated to showing how to construct $\alpha \in \mathbb{R}$ and its companion decoder function f_α. In order to do this, we need to lay out a couple of conceptual tools upon which the whole trick is based.

$\boxed{a)\ \textit{Fixed-point binary representation}}$ Without loss of generality, we focus on real numbers $\alpha \in [0, 1]$ in the unit interval. Since the integral part is identically null, the fractional part of any such α can be expressed as an infinite vector of coefficients $a_i = \{0, 1\}$ where each coefficient is paired with a weight $1/2^i$ as illustrated by:

$1/2^1$	$1/2^2$	$1/2^3$	$1/2^4$	$1/2^5$	$1/2^6$	$1/2^7$	$1/2^8$	$1/2^9$	$1/2^{10}$	$1/2^{11}$	$\cdots\cdots$

$\alpha = 0.$	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	$\cdots\cdots$

Converting this binary representation of α to its equivalent decimal counterpart is accomplished by evaluating the following infinite-sum expression:

$$\alpha = \sum_{n=1}^{+\infty} \frac{a_n}{2^n} \; ; \quad \alpha \in [0, 1]$$

Unfortunately, existing data warehouse solutions can only store a finite (however large) amount of information and cannot handle the infinite memory requirements imposed by mathematical real numbers in $\mathbb{R}$. Instead, one has to decide on a threshold to approximate real numbers thereby resulting in a finite τ-bit expansion:

Truncation to finite τ-bit precision ($\tau = 8$)

$0.$	0	0	0	0	0	0	0	0	$= 0$
$0.$	0	0	0	0	0	0	0	1	$= 0.00390625$

$\vdots$

$0.$	1	0	0	0	0	0	0	0	$= 0.5$

$\vdots$

$0.$	1	1	1	1	1	1	1	0	$= 0.9921875$
$0.$	1	1	1	1	1	1	1	1	$= 0.9960938$

Discretization of the unit interval into $2^{\tau=8} = 256$ uniformly spaced representable numbers.

Empirical data points are rounded off to the nearest available number in this fixed-point representation.

The equivalent decimal value is obtained by evaluating the now-finite sum:

$$\alpha_{\text{approx}} = \sum_{n=1}^{\tau} \frac{a_n}{2^n}$$

This approximation is completely faithful only in the degenerate case where all components $a_{n \geq \tau+1} \equiv 0$. As soon as one of these components is non-zero α_{approx} always underestimates the true value α. In the worst case scenario, all the neglected components are $a_{n \geq \tau+1} \equiv 1$ showing that the error is bounded by:

$$|\alpha - \alpha_{\text{approx}}| \leq \sum_{n=\tau+1}^{+\infty} \frac{1}{2^n} = \frac{1}{2^\tau}$$

This type of uniform discretization of the reals is referred to as a "fixed-point" representation to be contrasted with "floating-point" numbers that can represent both large and small reals in a reasonable amount of storage using a system similar to scientific notation. Let us mention, in passing, that with the surge of compute-intensive workloads based on neural networks, research towards reducing hardware overhead has also sparked a renewed wave of interest in alternative number representations such as the "logarithmic number system" [5]...
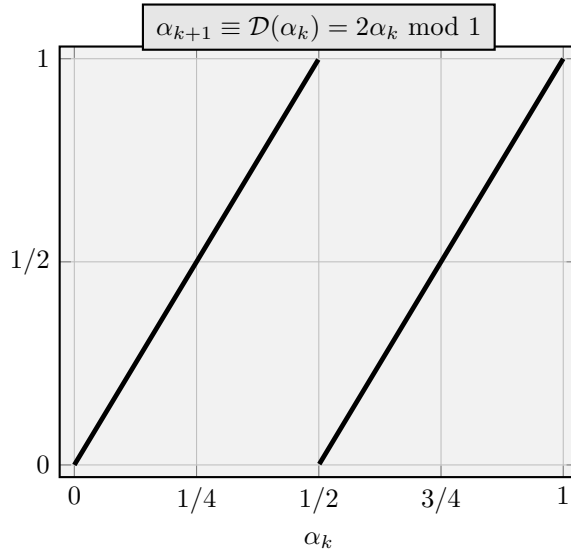
$$\alpha_{k+1} \equiv \mathcal{D}(\alpha_k) = 2\alpha_k \bmod 1$$

Figure 4: Graphical illustration of the dyadic transformation $\mathcal{D}$. Notice how the modulo operation ensures that $\alpha_{k+1} \in [0,1] \;\; \forall k \in \mathbb{N}$ by creating a non-differentiable jump at $1/2$ turning $\mathcal{D}$ into a piecewise linear function.

b) Chaos theory   The second (and last) prerequisite consists of a particularly simple incarnation of a one-dimensional discrete dynamical system known as the "dyadic transformation". Given a variable $\alpha_k \in [0,1]$ at time-step k, its evolution at time-step $k+1$ is defined by:

$$\alpha_{k+1} \equiv \mathcal{D}(\alpha_k) = 2\alpha_k \bmod 1 \tag{2}$$

Stepping away from decimal representation and moving over to binary form helps reveal the underlying behavior of $\mathcal{D}$ (illustrated graphically in Figure 4). Indeed, in this representation, multiplication by 2 can be interpreted as a simple shift of the whole bit sequence defining α_k by a single bit in the leftwise direction. In addition, the modulo operation ensures that every bit coming over on to the left-side of the radix point gets turned into a 0 thereby ensuring that α_{k+1} remains in the unit interval. Because of this property, the dyadic transformation is sometimes referred to as the "bit-shift" map. As an example, let us consider $\tau = 8$ significant bits and denote by ⬚ the infinite sequence of random bits that follow. Iterated applications of $\mathcal{D}$ can be understood as an accumulation of leftwise bit-shifts:
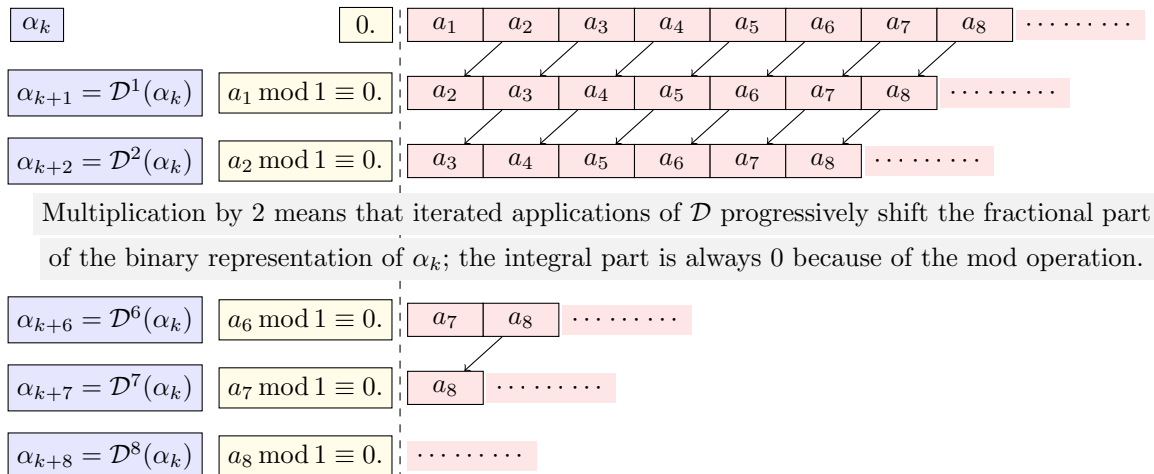
In addition to clarifying the bit-shift property of the dyadic transformation, this visualization nicely brings to light the fundamental mechanism responsible for the onset of chaotic dynamics: each iteration of $\mathcal{D}$ leads to the loss of 1 bit of information. After τ iterations all the significant bits of α_k have been lost and we are left with an infinite sequence of random bits. In other words, the time evolution of α_k depends very strongly on its least significant bits: a hallmark of chaos known as sensitivity to initial conditions.

5

Encoder/decoder strategies. Let us consider a dataset $\mathcal{X} = [x_0, \cdots, x_n]$ composed of a sequence of samples whose values have been normalized to lie in the unit interval. Any data source, regardless of the underlying dimensionality or modality, can be collapsed into such a list of numerical values and min-max normalized so that whatever pre-processing procedure one may need to apply to the raw data does not impose any restrictions on our approach (as demonstrated before by the illustrative examples for scatter plots, sound, images...).

Already, combining the fixed-point binary representation of real numbers along with the dyadic transformation allows us to construct a straightforward encoding/decoding strategy that is able to reproduce all the samples of $\mathcal{X}$ up to an arbitrary degree of precision using a single parameter in $\mathbb{R}$ (section 1). Because of its lack of sophistication, this strategy won't be able to explain the elegant expression f_α displayed at the very beginning of the paper. Nevertheless, it will serve as a conceptual scaffolding upon which our second strategy will be heavily based. Uncovering the origin of f_α will require the introduction a few more theoretical tools from chaos theory: logistic maps and the concept of topological conjugacy. Blending these new ingredients on top of the blueprint laid out in the first construction will finally reveal how the differentiable model f_α comes about (section 2).

1 Building the scaffolding

The idea is almost embarrassingly simple... Start by encoding all the values of $\mathcal{X}$ into a long binary string and convert it to its decimal representation. Use this number $\alpha_0 \in \mathbb{R}$ as the initial condition for the dyadic transformation. Because of its bit-shift property, iterated applications of $\mathcal{D}$ can be used to decode one-by-one all the components of $\mathcal{X}$.

1.1 Construction of the initial condition

We start by converting all the values $[x_0, x_1, x_2, \cdots, x_n]$ of $\mathcal{X}$ into their binary representation truncated to the first τ significant bits.

Convert all samples to fixed-point binary representation with τ bits of precision
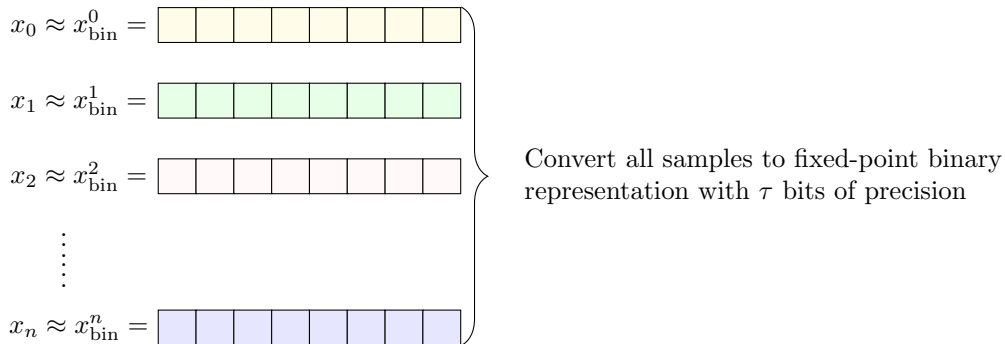
Next, we concatenate all the samples together so that the entire dataset is "flattened" into a long binary string composed of $\sim n\tau$ bits. Notice how we associate the binary coefficients with weights of decreasing significance according to their order of appearance in $\mathcal{X}$.
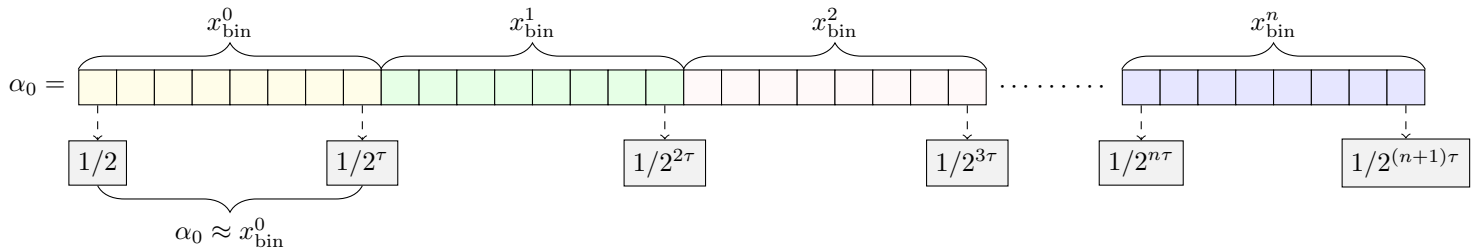
Thanks to this encoding scheme, the bits corresponding to x_{bin}^0 occupy the first τ dominant weights ensuring that α_0 is already a good approximation to the value of the first sample:

$$\alpha_0 \approx x_0 \quad \text{with error bound} \quad |\alpha_0 - x_0| < 1/2^\tau$$

The error bound is explained in the introductory paragraph on fixed-point binary representations.

1.2 Decimal precision requirements

Converting $\alpha_0 \in [0, 1]$ to its equivalent decimal representation is achieved by evaluating the expression:

$$\alpha_0 = \sum_{i=1}^{(n+1)\tau} \frac{\alpha_i}{2^i} = \left(\sum_{i=1}^{\tau} + \sum_{i=\tau+1}^{2\tau} + \sum_{n=2\tau+1}^{3\tau} + \cdots + \sum_{n=n\tau+1}^{(n+1)\tau} \right) \frac{\alpha_i}{2^i}$$

Conceptually, this summation can be partitioned into blocks which are color-coded according to the level of significance they represent in the original binary representation of α_0. Each block encodes the value x_j of a particular sample corrected by a multiplicative factor determined by the order in which sample j appears in the dataset $\mathcal{X} = [x_0, x_1, x_2, \cdots, x_n]$.

Since all further arithmetic operations will exclusively be performed in decimal representation, it is important to answer the following question: How many digits of decimal precision does one need to keep in order to have the capacity to represent all the information contained in the original construction of α_0 as a long binary string? Let us denote by p_{bin} the level of binary precision (number of bits) targeted. This means that we aim to describe numbers with a numerical precision of at least $1/2^{p_{\text{bin}}}$. Therefore the required level of decimal precision p_{dec} is determined by:

$$\frac{1}{10^{p_{\text{dec}}}} = \frac{1}{2^{p_{\text{bin}}}} \implies p_{\text{dec}} = \left(\frac{\log 2}{\log 10} \approx 0.3 \right) p_{\text{bin}}$$

Unsurprisingly, decimal representation requires less digits than would otherwise be necessary in binary to achieve the same level of numerical precision. Intuitively, this can be attributed to the fact the decimal representation of a real number can be expanded over a larger set of base digits $\{0, \cdots, 9\}$ instead of just $\{0, 1\}$ in binary form.

With typical numbers such as $n = 1000$ data points and an accuracy of $\tau = 8$ bits of precision for each sample, we need to faithfully encode $p_{\text{bin}} \approx 8000$ bits of information which corresponds to $p_{\text{dec}} \approx 2400$ significant decimal digits. Obviously, this level of precision goes way beyond the traditional IEEE floating-point standard and other common data types offered by host programming languages. Therefore, actual implementation of the algorithm requires the use of external libraries to perform arbitrary-precision arithmetic operations.

1.3 Decoding using the dyadic transformation

Going back to the definition of $\mathcal{D}$ in eq.(2), it is easy to see that multiple iterated applications of the dyadic transformation $k\tau$ times on the initial condition α_0 leads to a new value α_k which is given by:

$$\alpha_k \equiv \underbrace{(\mathcal{D} \circ \cdots \circ \mathcal{D})}_{k\tau \text{ times}} (\alpha_0) = \mathcal{D}^{k\tau}(\alpha_0) \implies \boxed{\alpha_k = 2^{k\tau} \alpha_0 \bmod 1 \quad ; \quad k \in \{0, \cdots, n\}} \tag{3}$$

As will be demonstrated below, this expression for α_k is indeed able to reproduce all the datapoints of $\mathcal{X}$.

With $k = 0$, we have seen that α_0 is already a good approximation to the first sample of our dataset. In order to see how the remaining datapoints can also be extracted using $\mathcal{D}$, it is helpful to mentally go back to the binary representation in which the dyadic transformation can be seen as a simple leftwise bit-shift. Note that this logic is only presented **as a visual guide** and that α's are calculated directly in decimal basis. Accordingly, let us start by looking at $k = 1$ so that the first iterate α_1 corresponds to:
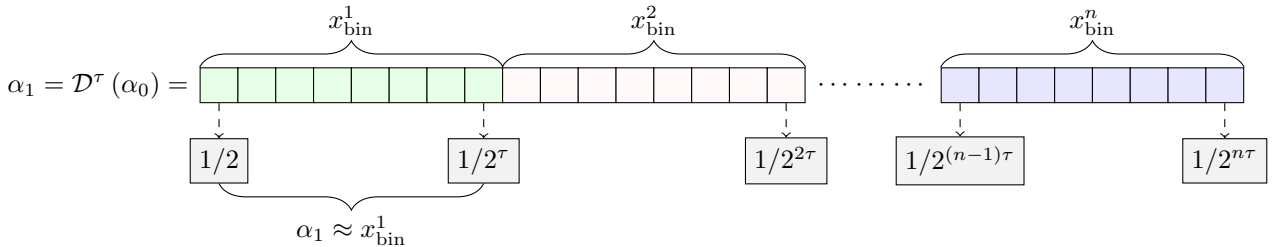
showing that the second sample is recovered by τ successive applications of $\mathcal{D}$ on our carefully crafted initial condition:

$$\alpha_1 = 2^{\tau} \alpha_0 \bmod 1 \implies \alpha_1 \approx x_1 \text{ with error bound } |\alpha_1 - x_1| < 1/2^{\tau}$$

Moving on to $k = 2$, we have:

$$\alpha_2 = \mathcal{D}^{2\tau}(\alpha_0) =$$
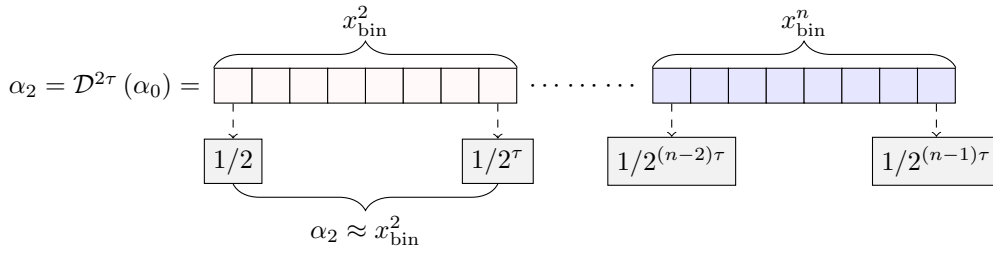
confirming that the third sample of $\mathcal{X}$ is indeed reproduced:

$$\alpha_2 = 2^{2\tau}\alpha_0 \bmod 1 \implies \alpha_2 \approx x_2 \text{ with error bound } |\alpha_2 - x_2| < 1/2^{\tau}$$

Generalizing, it is clear that iterating the dyadic transformation $n\tau$ times removes away all the bits of α_0 leaving only its (initally) most subdominant contribution (now dominant) which was constructed as the sequence of τ bits encoding the last sample:

$$\alpha_n = \mathcal{D}^{n\tau}(\alpha_{\text{bin}}) =$$
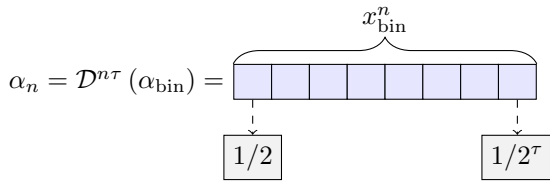
thereby completing our demonstration that the entire dataset $\mathcal{X}$ can be decoded by eq.(3):

$$\alpha_n = 2^{n\tau}\alpha_0 \bmod 1 \implies \alpha_n \approx x_n \text{ with error bound } |\alpha_n - x_n| < 1/2^{\tau}$$

1.4   Summary in code

Let us finish this section by showing how the algorithm described above can be implemented very easily in just a few lines of code. *(Helper functions to convert between binary and decimal representations, using a default value of $\tau = 8$, as well as the construction of a dataset $\mathcal{X}$ composed of 50 random numbers in the unit interval are provided in appendix A.)*

The first step consists in looping over the dataset to approximate the decimal values of all of its samples with their τ-bit binary representations. This transformation of $\mathcal{X} = [x_0, x_1, x_2, \cdots, x_n] \approx [x_{\text{bin}}^0, x_{\text{bin}}^1, x_{\text{bin}}^2, \cdots, x_{\text{bin}}^n]$ is achieved by using the helper function `decimalToBinary` (which internally invokes the dyadic transformation implemented as `dyadicMap` below). Appending these τ-long binary strings all together by order of appearance in $\mathcal{X}$ (denoted in code as `xs`) leads to the initial condition α_0 (denoted in code as `binaryInitial`).

```python
# xs: List[float] is the dataset instantiated here as a list of random values in [0, 1] as
    defined the Appendix.

dyadicMap = lambda x : (2 * x) % 1

binaryInitial = ''.join(map(decimalToBinary, xs))

print('binaryInitial = %s\n' % binaryInitial)

binaryInitial = 10001100101101111001101010001011011011001010010101110000111001001111101100110
                00101100101010000111100100011110110000010010000101100000010111010101111000111
                11011110111110101100110001110110110001110001111010100011001001001111000011000
                01010110101001000011110001100111010010010001000001001001111010011100100011101
                11110001101011100101110001101111101100100000011111010101010101011001101010010
                0001010100000101101
```

Since $\mathcal{X} \equiv$ `xs` is composed of 50 samples represented in binary with $\tau = 8$ bits of precision, we can verify that the initial condition `binaryInitial` requires $50 \times 8 = 400$ bits of precision.

Converting this initial condition from its binary form into its corresponding decimal representation is achieved by invoking the `binaryToDecimal` helper function.

```
1  decimalInitial = binaryToDecimal(binaryInitial)
2
3  print('decimalInitial = %s' % decimalInitial)
4
5  decimalInitial = 0.5496765699760055703169202362353308700123406974106022231112441515212066847
6                     39906068280491630119676272608712990165274607279 72
```

As can be verified from the complete value of $\alpha_0 \equiv$ `decimalInitial`, one needs to keep about ≈ 120 decimal digits of precision. Next, we implement the decoding function $\alpha_k \equiv$ `dyadicDecoder`.

```
1  dyadicDecoder = lambda k: (2 ** (k * tau) * decimalInitial) % 1
```

It is important to recognize that the multiplication between α_0 and the k-dependent exponential pre-factor should be understood in the context of "overloading". In particular, this means that the implementation is dispatched to an external library that handles arithmetic operations on arbitrary-precision objects such as `decimalInitial` (instead of being handled by the native implementations on common data types offered by the host programming language).

Now that we have constructed the initial condition and the decoding function, all the values of the original dataset $\mathcal{X}$ can be generated (within a level of precision determined by τ) one-by-one by applying `dyadicDecoder` on a range of integer-valued arguments $k \in [0, \cdots, n]$. Even though arbitrary-precision numbers were necessary to carry out accurate arithmetic operations on `decimalInitial`, the components of the resulting list denoted as `decodedValues` can be downcast to usual floating-point numbers using the built-in `float` method.

```
1  n = len(xs)
2  decodedValues = [float(dyadicDecoder(_)) for _ in range(n)]
```

As a sanity check, it is interesting to check that the difference between the values $[\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_n]$ produced by our encoding/decoding strategy and the true values from $\mathcal{X} = [x_0, x_1, x_2, \cdots, x_n]$ satisfy the theoretical error bound $|\alpha_j - x_j| < 1/2^\tau$ captured by `normalizedErrors` in the code fragment below.

```
1  maxError = 1 / 2 ** tau
2  normalizedErrors = [abs(decodedValue − dataPoint) / maxError
3                        for decodedValue, dataPoint in zip(decodedValues, xs)]
```
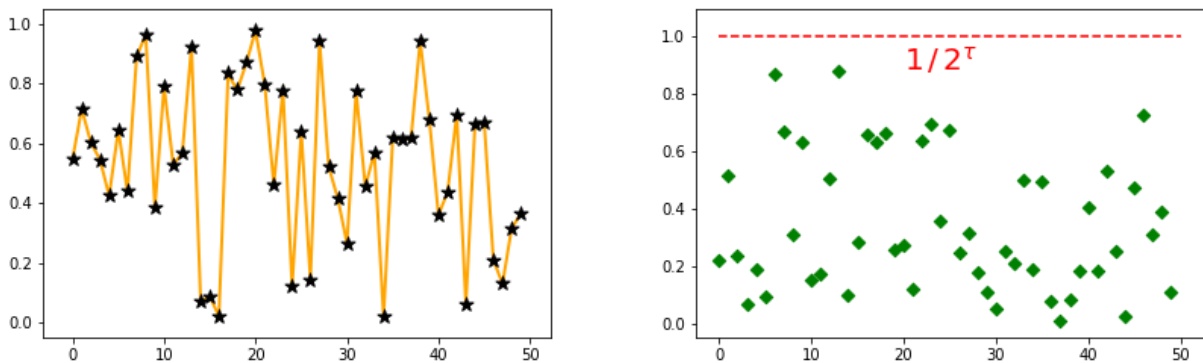

Figure 5: **Left)** Comparison between the decoded values (black star markers) against the original dataset $\mathcal{X}$ (thick orange line). **Right)** Normalized error showing that the gap between decoded values and the original values never strays more than $1/2^\tau$ away (represented as the red horizontal dashed line) confirming the error bound derived in the main text.

2   Applying makeup to the scaffolding

We refer to the strategy described in the previous section as *"building the scaffolding"*. The reason for this choice of terminology is that it feels like the decoding function leaves too much of the bare mechanics exposed. Most objectionable, it relies in plain view on the dyadic transformation and by implication on its bit-shift property. Moreover, the somewhat inelegant modulo operation with its lack of continuity may stand out as a "hair in the soup" in an age where **differentiable programming** is all the rage.

The purpose of this section is to justify the more aesthetically pleasing formulation $f_\alpha(x) = \sin^2(2^{x\tau} \arcsin \sqrt{\alpha})$ whose accomplishments have already been documented in the introduction as a replacement to the crude decoding function, eq.(3), that we currently have. To do this, we need to introduce two well-known concepts in the context of dynamical systems: the logistic map and the notion of topological conjugacy. It is worth emphasizing that these ingredients will be used simply as a kind of "makeup" on top of an otherwise mostly identical logic to the one presented in the previous section.

2.1   Logistic map

Originally introduced as a discrete-time model for population demographics, the logistic map $\mathcal{L}$ is a polynomial mapping between a real-valued $z_k \in [0, 1]$ and its next iterate z_{k+1} defined by:

$$z_{k+1} \equiv \mathcal{L}(z_k) = r\, z_k(1 - z_k) \tag{4}$$

Note that we will restrict ourselves to the special case where $r = 4$. Even if not particularly useful in biology, it turns out that the logistic map exhibits an unexpected degree of complexity [6]. In fact, it is often used nowadays as the archetypal example of how chaotic behavior may arise from simple dynamical equations.

A quick visual comparison of $\mathcal{L}$ (as depicted in Figure 6) with the dyadic transformation $\mathcal{D}$ (illustrated previously in Figure 4) suggests that both maps do not have much in common with each other... For example, it is clear that $\mathcal{L}$ is continuous and differentiable over the unit interval; smoothness guarantees that stand in stark contrast to the sawtooth shape associated with $\mathcal{D}$.

At this point you may ask: well, what does this logistic map business have to do with anything we discussed so far?
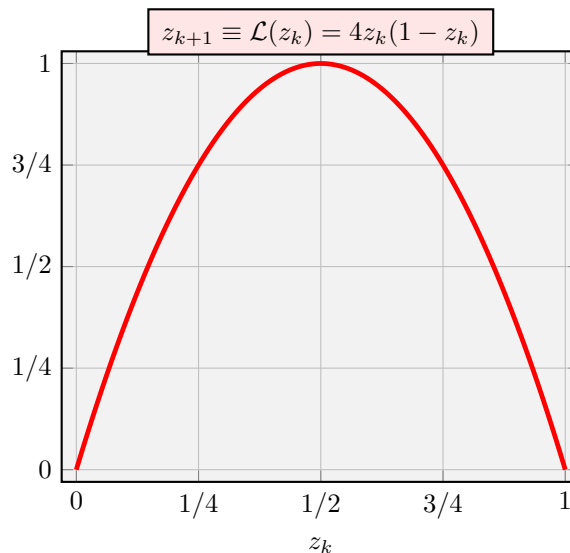

Figure 6: Graphical illustration of the logistic map $\mathcal{L}$ as defined in eq.(4) in the special case where $r = 4$. To be compared with the non-differentiable dyadic transformation $\mathcal{D}$ shown in Figure 4.
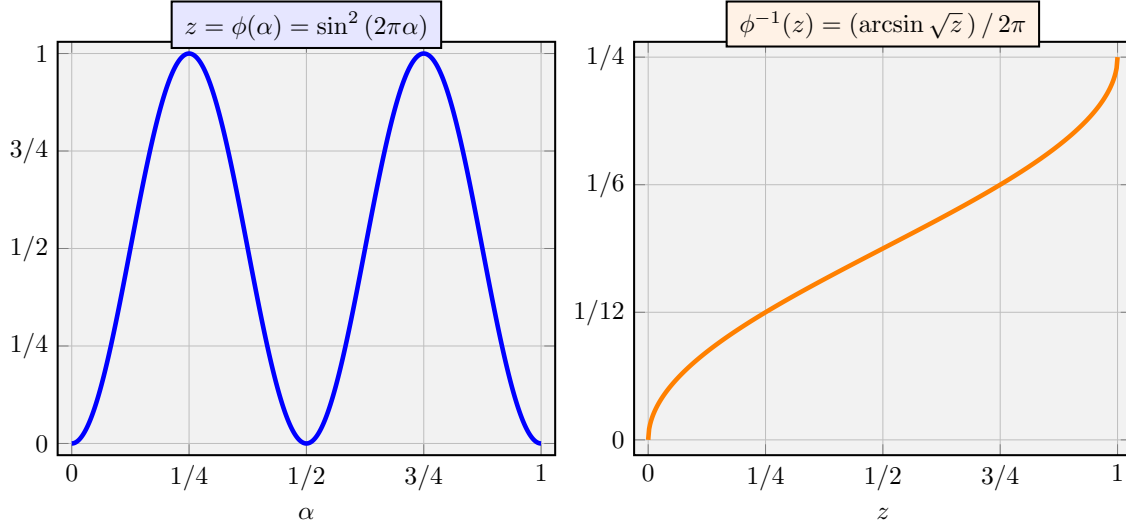
Figure 7: Illustrations of the homeomorphism ϕ and of its inverse ϕ^{-1}. Note that a continuous function with a continuous inverse function, as is the case here, is called a homeomorphism.

2.2 Toplogical conjugacy

Let us assume that for a given $k \in \mathbb{N}$ the value z_k of an iterate produced by the logistic map can be linked to some other variable α_k through the following change of variable:

$$z_k \equiv \phi(\alpha_k) = \sin^2(2\pi\alpha_k) \qquad (5)$$

Obviously, we will soon try and identify α_k as an iterate of the dyadic transformation $\mathcal{D}$ defined in eq.(2). For now, let us continue by following the evolution of z_k as prescribed by the logistic map. Using the newly-introduced ϕ, the value of the next iterate $z_{k+1} \equiv \mathcal{L}(z_k)$ can be expressed as:

$$z_{k+1} \equiv 4\phi(\alpha_k)\big(1 - \phi(\alpha_k)\big) = 4\sin^2(2\pi\alpha_k)\cos^2(2\pi\alpha_k) \implies z_{k+1} = \sin^2(2\pi\, 2\alpha_k)$$

In order to make a connection, let us recognize that:

$$\sin^2\big(2\pi\mathcal{D}(\alpha_k)\big) = \begin{cases} \sin^2(2\pi 2\alpha_k) & (\alpha_k < 1/2) \\ \sin^2\big(2\pi(2\alpha_k - 1)\big) = \sin^2(2\pi 2\alpha_k - 2\pi) = \sin^2(2\pi 2\alpha_k) & (\alpha_k > 1/2) \end{cases}$$

due to the periodicity of ϕ as one can see in Figure 7. Plugging this result back into the expression for z_{k+1} and introducing $\alpha_{k+1} \equiv \mathcal{D}(\alpha_k)$ leads to the following identity:

$$z_{k+1} = \phi(\alpha_{k+1}) \quad \Leftrightarrow \quad \mathcal{L}(z_k) = \phi\big(\mathcal{D}(\alpha_k)\big)$$

This result is nothing short of astonishing: it proves that although $\mathcal{L}$ and $\mathcal{D}$ superficially look like very different recurrence relations, it is more appropriate to think of them as topological twins that share identical dynamics. With ϕ and its inverse ϕ^{-1} acting as a bridges between the two topological spaces, this marvelous behavior known as topological conjugacy can be summarized pictorially via the commutative diagram:
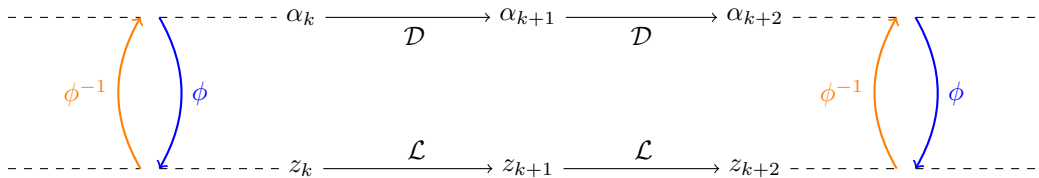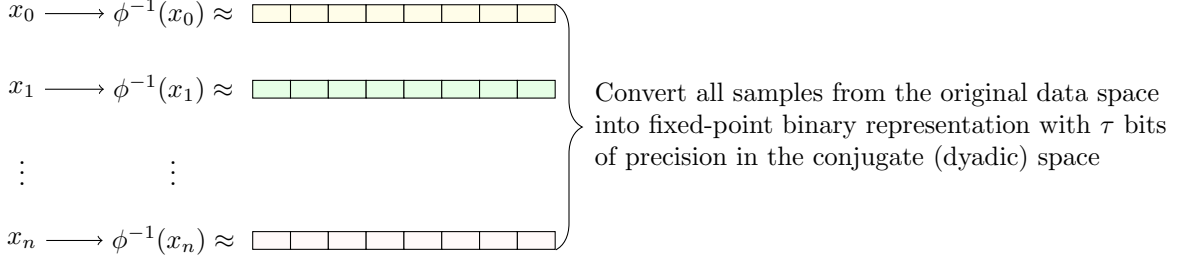

In other words, one may decide to study the dynamics of a variable according to the logistic map and never leave this representation but, all the while, think about how this dynamics is equivalently represented in some kind of "parallel universe" where variables evolve according to the dyadic transformation up to a compositional "portal" in the form of ϕ.

Now that we have this remarkable phenomenon in our hands, how can we exploit it to apply makeup to the basic encoding/decoding strategy presented in the previous section?

2.3 Construction of the initial condition

Instead of populating the initial condition with direct binary approximations to the components of the original dataset, let us start with a pre-processing step by applying ϕ^{-1} to all of the samples of $\mathcal{X} = [x_0, x_1, x_2, \cdots, x_n]$ before going into a fixed-point binary representation:

Convert all samples from the original data space into fixed-point binary representation with τ bits of precision in the conjugate (dyadic) space

Concatenating all these transformed data points into a long binary string composed of $\sim n\tau$ bits associated with weight factors of decreasing significance factor (see section 1.1) leads to the following initial condition:
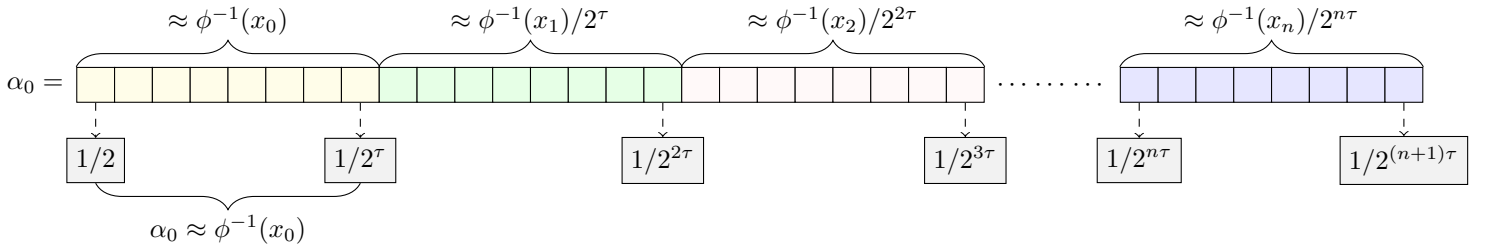
which we immediately convert to its corresponding decimal form via $\alpha_0 = \sum_{i=1}^{(n+1)\tau} \alpha_i / 2^i$ keeping in mind the required number of digits of precision (see section 1.2). Having built α_0 in the conjugate (dyadic) space, it is now time to use topological conjugacy in order to transport this initial condition back out into the original data domain by applying the ϕ "portal". In other words, we consider an initial condition z_0 defined as:

$$z_0 = \phi(\alpha_0) = \sin^2(2\pi\alpha_0)$$

Because of our careful construction of the first τ significant bits leading to $\alpha_0 \approx \phi^{-1}(x_0)$ and the fact that ϕ is a homeomorphism, we see that $z_0 = \phi(\alpha_0) \approx x_0$ is a good approximation to the first sample:

$$z_0 \approx x_0 \ \text{ with error bound } |z_0 - x_0| < \pi/2^{\tau-1}$$

The error bound between the generated value z_0 and the true value of the first sample x_0 of $\mathcal{X}$ will be explained later in section 2.5 but is already reported here for the sake of completeness.

2.4 Decoding using the logistic map

Thanks to the topological conjugacy relationship between the dyadic transformation and the logistic map, we can plug in the solution $\alpha_k \equiv \mathcal{D}^{k\tau}(\alpha_0) = 2^{k\tau}\alpha_0 \bmod 1$ obtained previously in the dyadic space, see eq.(3), into its conjugate counterpart $z_k \equiv \mathcal{L}^{k\tau}(z_0) = \phi(\alpha_k) = \phi(2^{k\tau}\alpha_0)$. Note that because the codomain of ϕ is restricted to the unit interval, the modulo operation initially present in α_k is not relevant anymore for z_k. Finally, we relate the initial condition α_0 constructed above (converted to decimal representation) to its conjugate value z_0 via $\alpha_0 = \phi^{-1}(z_0) = (\arcsin\sqrt{z_0})/2\pi$. All together, the iterated evolution of the logistic map admits an exact solution 😊😊😊 **functionally identical to** f_α 😊😊😊 our promised crown jewel that was prophesied on the first page of this paper. Continuing with the notation used here, this decoding function is given by:

$$z_k = \sin^2\left(2^{k\tau}\arcsin\sqrt{z_0}\right) \tag{6}$$

We have already seen that the first sample is well approximated with $z_0 \approx x_0$. Because of topological conjugacy, we will now show that all other samples can also be recovered by sampling z_k at $k \in [0, \cdots, n]$.

To gain more insight into why the dynamics of the logistic map reproduces the remaining samples of $\mathcal{X}$, let us consider the first iteration of z_0 under $\mathcal{L}$. Following the "parallel universe" interpretation of topological conjugacy depicted in the diagram of paragraph 2.2, we see that the first iterate $z_1 = \mathcal{L}^\tau(z_0)$ is conjugate to $\alpha_1 = \mathcal{D}^\tau(\alpha_0)$. In this conjugate (dyadic) space, the combined effect of τ successive iterations of $\mathcal{D}$ results in a leftwise block shift of the first τ bits of the initial condition α_0. Effectively, the bit sequence that was originally encoding $\phi^{-1}(x_0)$ is discarded (moved over the binary point and turned to 0 by the modulo operation) making room for the next sequence of τ bits encoding $\phi^{-1}(x_1)$ to become the leading contribution to the numerical value of α_1.
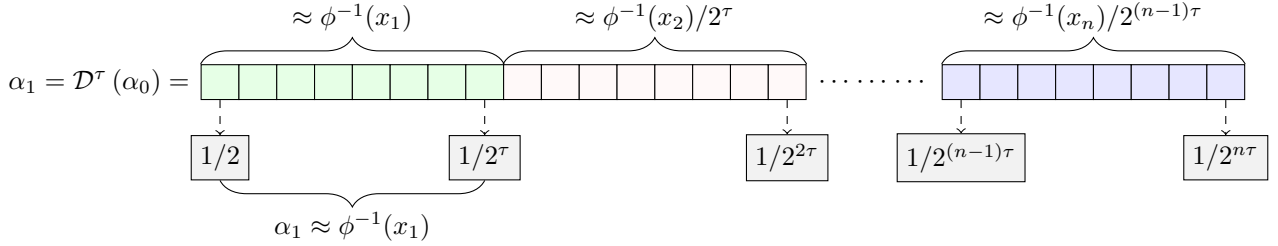
Thanks to our portal between topological spaces, the data space sample z_1 is conjugate to $\alpha_1 \approx \phi^{-1}(x_1)$ meaning that $z_1 = \phi(\alpha_1) \approx x_1$. This result confirms that appying eq.(6) with $k = 1$ leads to:

$$z_1 = \sin^2\left(2^\tau \arcsin\sqrt{z_0}\right) \approx x_1 \quad \text{with error bound} \quad |z_1 - x_1| < \pi/2^{\tau-1}$$

The error bound will be explained in the next paragraph. Moving on to the second iteration, $z_2 = \mathcal{L}^{2\tau}(z_0)$ is conjugate to $\alpha_2 = \mathcal{D}^{2\tau}(\alpha_0)$ which is represented in the conjugate space by:
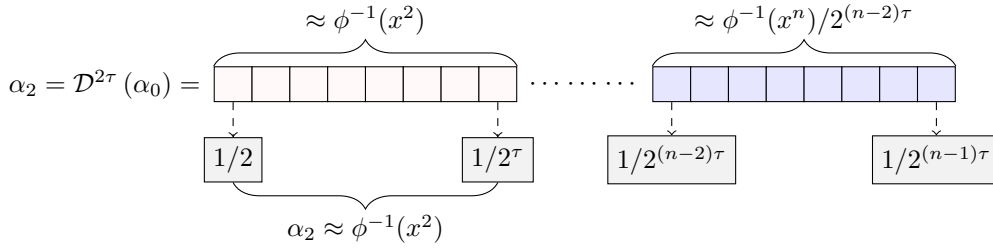
Once again, invoking topological conjugacy arguments to the observation that $\alpha_2 \approx \phi^{-1}(x_2)$ confirms that the third sample is indeed recovered $z_2 = \phi(\alpha_2) \approx x_2$ by sampling eq.(6) at $k = 2$. In other words:

$$z_2 = \sin^2\left(2^{2\tau} \arcsin\sqrt{z_0}\right) \approx x_2 \quad \text{with error bound} \quad |z_2 - x_2| < \pi/2^{\tau-1}$$

Pushing this logic n times, it is clear that the binary representation of α_0 in the conjugate space gets gradually stripped off of all of its leading components finally leaving only the bits corresponding to the last transformed data point of $\mathcal{X}$. In picture:
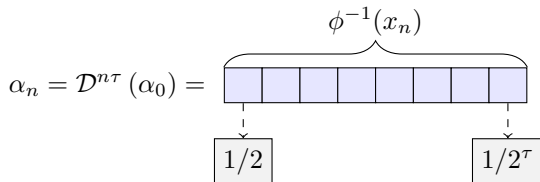
confirming that $\phi^{-1}(x_n)$, originally constructed as the least signifiant τ bits of the initial condition α_0, becomes the leading contribution to α_n. Therefore $z_n = \phi(\alpha_n) \approx x_n$ is correctly decoded:

$$z_n = \sin^2\left(2^{n\tau} \arcsin\sqrt{z_0}\right) \approx x_n \quad \text{with error bound} \quad |z_n - x_n| < \pi/2^{\tau-1}$$

Before moving on to a detailed explanation of the error bound, let us stress how far we have gone: the entire dataset $\mathcal{X}$ has been reproduced by the beautiful f_α as promised in the introduction!

2.5 Error bound: Lipschitz continuity

In order to complete the proof, let us justify the error bound $|z_j - x_j| < \pi/2^{\tau-1}$ between a sample x_j drawn from the dataset $\mathcal{X}$ and its corresponding approximation $z_j = \sin^2\left(2^{j\tau} \arcsin\sqrt{z_0}\right) \approx x_j$ given by eq.(6) (which is identical to f_α shown in the introduction).

According to the mean value theorem, given a function ϕ differentiable over an interval (a, b), there exists a point $\xi \in (a, b)$ in this interval such that $\phi'(\xi) = (\phi(a) - \phi(b))/(a - b)$. Separately, it is easy to verify that the homeomorphism ϕ, our portal between different topological spaces defined in eq.(5), is Lipschitz continuous: its derivative is bounded by $|\phi'(\xi)| = 2\pi |\sin 4\pi\alpha| \leq 2\pi$. Plugging this Lipschitz constant of 2π into the expression of the mean value theorem leads to:

$$|\phi(a) - \phi(b)| \leq 2\pi |a - b|$$

We can now specialize this result to the case where $a = \alpha_j$ and $b = \phi^{-1}(x_j)$. Thanks to the topological conjugacy relationship $z_j = \phi(\alpha_j)$ and the inequality above, we get:

$$|z_j - x_j| \leq 2\pi \left|\alpha_j - \phi^{-1}(x_j)\right|$$

Because $\phi^{-1}(x_j)$ weighs in as the leading contribution to the numerical value of α_j (up to its first τ bits), we have $\alpha_j \approx \phi^{-1}(x_j)$ with an error bound $\left|\alpha_j - \phi^{-1}(x_j)\right| < 1/2^\tau$ (see section 1). Combining this inequality together with the previous one derived above leads to the promised error bound:

$$\boxed{|z_j - x_j| < \frac{\pi}{2^{\tau-1}}}$$

This shows that using the logistic map to decode the dataset generates a degradation by a factor of 2π compared to what can be achieved with the more straightforward strategy described in section 1.

2.6 Summary in code

Inspired by the previous section, we conclude by showing how the algorithm can easily be implemented in just a few lines of code. *(Helper functions to convert between binary and decimal representations, using a default value of $\tau = 8$, as well as the construction of a dataset $\mathcal{X}$ composed of 50 random numbers in the unit interval are provided in appendix A.)*

We start by applying ϕ^{-1} (denoted by `phiInv`) to all the samples of $\mathcal{X}$ and follow by expressing the resulting values as fixed-point precision (defaulted to τ bits) binary numbers. Concatenating all these transformed samples together yields the conjugate initial value α_0 in binary.

```
1  # xs: List[float] is the dataset instantiated here as a list of random samples with decimal
2  #     values defined the Appendix
3
4  phiInv                  = lambda z: np.arcsin(np.sqrt(z)) / (2 * np.pi)
5  decimalToBinary_phiInv  = lambda z: decimalToBinary(phiInv(z))
6
7  conjugateInitial_binary = ''.join(map(decimalToBinary_phiInv, xs))
8  print('conjugateInitial_binary = %s\n' % conjugateInitial_binary)
9
10 conjugateInitial_binary = 0010000100101001001001000010000100011100001001100001110100110010000
11                           1110000011011001011000010000100100010001101000000101000001100000
12                           0101001011100010110000110000001110100010110100011110001011000000 11
13                           1000100101000011110011011000100000000111000001011000101011000111 10
14                           0010001000000101001001000010010000100100001101100010011100011010 00
15                           0111010010100000001010001001100010011100010011000011100001100000 01
16                           1010
17
18 conjugateInitial = binaryToDecimal(conjugateInitial_binary)
19 print('conjugateInitial = %s' % conjugateInitial)
20
21 conjugateInitial = 0.12953401382778691458786695916416542476624903080900276738757903052119237
22                    70384582661092902913534541786594536762369093523 7
```

As before, the 50 samples require $50 \times (\tau = 8) = 400$ bits of precision which translates to ≈ 120 digits of decimal precision as can be seen respectively in `conjugateInitial_binary` and `conjugateInitial`.

One peculiarity of having used the ϕ^{-1} function is that every τ^{th} bit of `conjugateInitial_binary` is 0. This is because $\phi^{-1}(z) < 1/4$, $\forall z \in [0,1]$ as can be gleaned from Figure 7. Since leading bits are associated with a weight $1/2$, the first non-zero bit for real numbers $< 1/4$ is always the second bit. Although this introduces a detectable redundancy which causes additional $\approx 10\%$ memory requirements, more efficient strategies come at the cost of extra complexity that we do not wish to explore here.

The next step consists in transporting the decimal version of the initial condition $\alpha_0 \equiv$ `conjugateInitial` from the conjugate space to the original data space by using the ϕ transformation:

```
from mpmath import pi as Pi
from mpmath import sin as Sin

# Note that (...) ** 2 is also overloaded and dispatches to mpmath.power(..., 2)

phi = lambda alpha: Sin(2 * Pi * alpha) ** 2

decimalInitial = phi(conjugateInitial)
print('decimalInitial = %s' % decimalInitial)

decimalInitial = 0.52847263822305822321414776131142334134424126841548996094257891002142562165
                 61795491407103017116363770315500096396253164 2
```

As discussed in the code summary of the previous section, all arithmetic operations on $z_0 \equiv$ `decimalInitial` are overloaded and dispatched to an external library that handles arbitrary-precision operations. Finally, we implement the decoding function `logisticDecoder` equivalent to the original f_α that was obtained in eq.(6). Sampling at $k = [0, \cdots, n]$ builds a list `decodedValues` that approximates the original dataset $\mathcal{X}$.

```
from mpmath import asin as Asin
from mpmath import sqrt as Sqrt

logisticDecoder = lambda k: Sin(2 ** (k * tau) * Asin(Sqrt(decimalInitial))) ** 2
decodedValues = [float(logisticDecoder(_)) for _ in range(len(xs))]
```

Finally, one may verify that the theoretical error bound of $\pi/2^{\tau-1}$ is indeed satisfied by all the datapoints by making sure that `normalizedErrors` always stays below unity.

```
maxError = Pi / 2 ** (tau - 1)

normalizedErrors = [abs(decodedValue - dataPoint) / maxError
                    for decodedValue, dataPoint in zip(decodedValues, xs)]
```
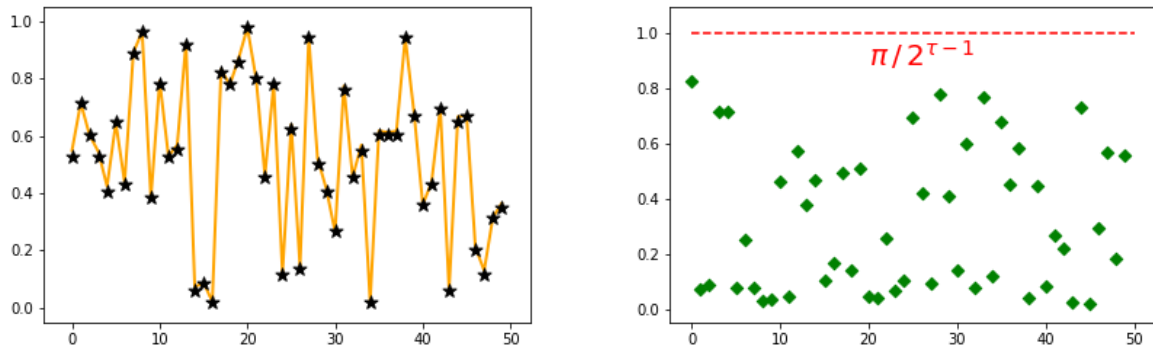

Figure 8: **Left)** Comparison between the decoded values (black star markers) against the original dataset $\mathcal{X}$ (thick orange line). **Right)** Normalized error showing that the gap between decoded values and the original values never strays more than $\pi/2^{\tau-1}$ away (represented as the red horizontal dashed line) confirming the error bound derived in the main text. Notice that the error bound is now 2π times larger than the one obtained using the more "naïve" encoding/decoding scheme of section 1.

3 Data science perspective: what about generalization?

How to think about our wonderful $f_\alpha(x) = \sin^2\left(2^{x\tau} \arcsin \sqrt{\alpha}\right)$ from the perspective of machine learning? This simple and well-behaved (continuous, differentiable...) function can be "trained" to reproduce any dataset $\mathcal{X}_{\text{train}}$ (up to an arbitrary level of precision determined by $\tau \in \mathbb{N}$) by appropriately adjusting a single parameter $\alpha \in \mathbb{R}$ "learned" from the training data. As a data scientist, an instinctive follow-up question would touch upon the generalization capabilities such a model may have on a previously unseen test dataset $\mathcal{X}_{\text{test}}$.

This is where the similarity with machine learning ends... As should be clear by now, the parameter α is not really learned from the training data as much as it is explicitly built as an alternative (encoded) representation of $\mathcal{X}_{\text{train}}$ itself. In other words, the information content of α is identical (i.e. no compression besides a tunable recovery accuracy determined by τ) to that of $\mathcal{X}_{\text{train}}$ and the model serves as nothing more than a mechanistic decoder. Building on the previous visualizations, α should be thought of pictorially as:

$$\alpha = 0.\left(\begin{array}{c}\mathcal{X}_{\text{train}}\end{array}\right)\left(\begin{array}{c}\cdots\cdots\cdots\end{array}\right)$$

where the encoded version of the training data is followed by an (infinite) series of random digits. In a way, decoding $\mathcal{X}_{\text{train}}$ can be seen as probing α at increasing levels of precision. Once all the components of $\mathcal{X}_{\text{train}}$ (green ellipsis above) have been extracted, we are starting to look at α at a level of precision for which the digits are now random (red ellipsis above). Despite providing an excellent fit to $\mathcal{X}_{\text{train}}$, there is no reason to expect f_α to provide any kind of generalization to data outside of $\mathcal{X}_{\text{train}}$ as demonstrated in Figure 9.

As already hinted at in the introduction, f_α puts in perspective the common practice of measuring the complexity of a machine learning model by counting its total number of parameters and suggests that other techniques based on data compression (such as minimum description length...) may be more appropriate. In addition, let us mention that given the huge number of parameters (each encoded as a floating-point number) available in modern deep learning architectures and the ease with which those models are able to fit randomly labeled data [7, 8], one falls back to a timely and often asked question in machine learning research: How much can the values of the weights of a well-trained neural network be attributed to brute-force memorization of the training data vs. meaningful learning and why do they provide any kind of generalization?
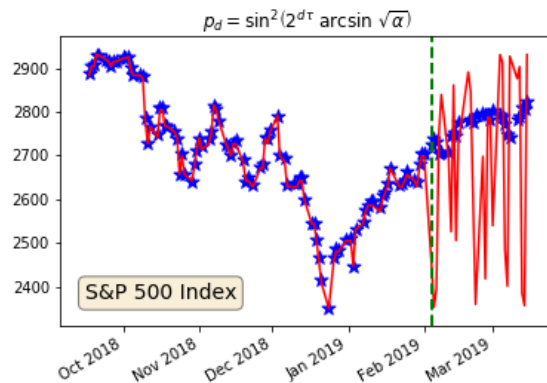

Figure 9: Comparison between the daily price values of the S&P 500 Index (blue markers) against the values predicted by $p_d = \sin^2\left(2^{d\tau} \arcsin \sqrt{\alpha}\right)$ parametrized by:

$\alpha = 0.\big($ 9186525008673170697061215177743819472103574383504939864690954692792184358812098296063847 31739470802166549191011747211905687147014341039869287275246189278502982951415770973892328 89947668652165705366720994855741788842509897413431 $\big)$ $\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$

where ≈ 230 decimal digits (equivalent to ≈ 760 significant bits in binary) encoding the first 95 daily price values ($95 = 760/\tau$ with default fixed-point precision $\tau = 8$ for each sample) are all explicitly provided. If one continues to apply p_d beyond the information contained in the significant digits of α, as delimited by the vertical dashed green line, the predictions start to take on random values within the range seen in the "training" phase.

16

Appendix A Helper functions

The "summary in code" paragraphs of sections 1 and 2 invoke a number of helper functions needed to perform common tasks such as i) converting between decimal/binary representations and ii) generating a playground dataset $\mathcal{X}$ composed of random values. All code fragments throughout the paper are written in the Python programming language using the `mpmath` library for arbitrary-precision arithmetic operations.

Converting between base 10 and base 2 representations. This is accomplished by `decimalToBinary` and `binaryToDecimal` that are both implemented below as fold expressions. (In addition we set $\tau \in \mathbb{N}$, as a working example, with a default value of $\tau = 8$.)

```
1  from functools import reduce
2  from mpmath import mp
3
4  tau = 8
5
6  def decimalToBinary(decimalInitial, targetBinaryPrecision = tau):
7      return reduce(lambda acc, _: [dyadicMap(acc[0]), acc[1] + ('0' if acc[0] < 0.5 else '1')],
8                    range(targetBinaryPrecision),
9                    [decimalInitial, ''])[1]
10
11 def binaryToDecimal(binaryInitial):
12     return reduce(lambda acc, val: acc + int(val[1]) / mpmath.power(2, (val[0] + 1)),
13                   enumerate(binaryInitial),
14                   mp.mpf(0.0))
```

where `dyadicMap`, the implementation of the dyadic transformation defined in eq.(2), can be found in the code summary paragraph of section 1.

The precision of the binary strings returned by `decimalToBinary` is kept to a small number τ of significant bits; as a result, this conversion can be implemented using native data types. On the other hand, the need to go from binary to decimal representations comes from the conversion of the initial condition (constructed as a very long binary string composed of $\sim n\tau$ significant bits encoding $\sim n$ datapoints) to an accurate decimal value. Keeping this level of precision when moving over to decimal representation requires the use of arbitrary-precision arithmetic operations (see section 1.2) provided by `mpmath` methods in `binaryToDecimal`.

Defining a random dataset $\mathcal{X}$. In order to give concrete results to the code fragments, let us consider a dataset $\mathcal{X}$ denoted as `xs` made up of 50 random values in the unit interval.

```
1  import numpy as np
2  np.random.seed(0)
3
4  numbDataPoints = 50
5  xs = np.random.uniform(0, 1, numbDataPoints)
6  print('xs = %s' % xs)
7
8  xs = [0.5488135  0.71518937 0.60276338 0.54488318 0.4236548  0.64589411
9   0.43758721 0.891773   0.96366276 0.38344152 0.79172504 0.52889492
10  0.56804456 0.92559664 0.07103606 0.0871293  0.0202184  0.83261985
11  0.77815675 0.87001215 0.97861834 0.79915856 0.46147936 0.78052918
12  0.11827443 0.63992102 0.14335329 0.94466892 0.52184832 0.41466194
13  0.26455561 0.77423369 0.45615033 0.56843395 0.0187898  0.6176355
14  0.61209572 0.616934   0.94374808 0.6818203  0.3595079  0.43703195
15  0.6976312  0.06022547 0.66676672 0.67063787 0.21038256 0.1289263
16  0.31542835 0.36371077]
```

References

[1] Steven Piantadosi. "One parameter is always enough". AIP Advances (2018) (U. C. Berkeley)

[2] *"I'm not very impressed with what you've been doing."* As recounted the famous physicist Freeman Dyson himself, this is how Nobel laureate Enrico Fermi started their 1953 meeting. *"Well, what do you think of the numerical agreement?"*, Dyson countered. To which, Fermi replied *"You know, Johnny von Neumann always used to say: With four parameters I can fit an elephant, and with five I can make him wiggle his trunk. So I don't find the numerical agreement impressive either."* (Life Stories of Remarkable People)

[3] Gregory Chaitin. "How real are real numbers?" (2011) (IBM Research)

[4] Source code to reproduce all the plots `https://github.com/Ranlot/single-parameter-fit` Executing the notebooks will also provide the complete values of parameter α that may extend to tens of thousands of digits. *(Apart from the code summary sections, the parameter α is presented in its entirety only in section 3 for the time series example.)*

[5] Jeff Johnson. "Rethinking floating point for deep learning". NeurIPS 2018 (Facebook AI Research)

[6] Robert May. "Simple mathematical models with very complicated dynamics." Nature-1976 (University of Oxford & Princeton University)

[7] "Understanding deep learning requires rethinking generalization". ICLR 2017 (MIT, U. C. Berkeley, Google Brain & Deep Mind)

[8] Congzheng Song, Thomas Ristenpart & Vitaly Shmatikov. "Machine Learning Models that Remember Too Much". ACM CCS 2017 (Cornell Tech)