Iterated-Integral Signatures in Machine Learning

by

Jeremy Francis Reizenstein

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Complexity Science and Statistics

April 2019

Contents

List of Tables

List of Figures

Acknowledgments

Thank you very much to my supervisors Anastasia Papavasiliou and Benjamin Graham. Thank you to so many people for collaborations, conversations and advice, especially Joscha Diehl, Terry Lyons, Sina Nejad, Harald Oberhauser, Michael Pearce, Daniel Wilson-Nunn, and the Complexity crew; and to many for broader support including my parents.

Thank you for the support of the Complexity Science Doctoral Training Centre, The Alan Turing Institute[1] and the Engineering and Physical Sciences Research Council, and to their staff. Computing resources for the work in section 5.8 were provided through a kind Microsoft Azure for Research award to the Alan Turing Institute. Thank you for helpful pointers from two anonymous reviewers.

[1]EPSRC grant EP/N510129/1

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The material in chapter 2 and chapter 3 stems from a collaboration with Joscha Diehl at MPI Leipzig. Most of sections 2.1, 2.2 and 2.3 appear in [DR18] which we have published. Sections 4.1 through 4.8, as well as parts of the introduction, appear in [RG18] with my original supervisor Ben Graham which has been submitted for publication. section 5.8 stems from a collaboration with Harald Oberhauser.

Abstract

The iterated-integral signature, or rough-path signature, of a path has proved useful in several machine learning applications in the last few years. This work is extended in a number of ways. Algorithms for computing the signature and log signature efficiently are investigated and evaluated, which is useful for many applications of signatures when working with large datasets. Online Chinese character recognition using signature features with recurrent neural networks is investigated. A recurrent neural network cell which stores its memory as the signature of a path is suggested and demonstrated on a toy problem.

There is an essentially unique element of the signature of a path in space which, under transformations of the space, scales with volume. That element is characterised geometrically.

Given two features of curves, you can make a new one by taking the signed area of the 2d curve those two features make as a curve is traced out. A simple algebraic description of those features (which turn out to be signature elements) which can be formed from linear combinations of such combinations of total displacements is conjectured and worked towards. This is know as "areas of areas".

Where is the code?

There has, naturally, been a lot of computer code involved in this project. The good algorithms for signature and log signature calculations in chapter 4 have been released as part of the Python library `iisignature`, whose homepage is at `github.com/bottler/iisignature`. From there, it should be easy to find its documentation and tests. Some associated tools etc. are in the `examples` subfolder. Several other parts of this project have companion code in repositories of `github.com/bottler`, for example at `https://github.com/bottler/phd-code` and `https://github.com/bottler/free-lie-algebra-py`.

Chapter 1

Introduction

The signature is an object which is crucial in the mathematical theory of rough paths[Lyo98], and the calculations have proved to be useful in machine-learning applications, particularly classification problems where the data itself is a stream or a path in space, ranging from an application to online Chinese handwriting recognition in 2013 [Gra13] to skeleton-based human action recognition in 2017 [Yan+17]. Other domains where the data has this form include signals from EEG and other medical monitors, sound and financial time series, where some set of numbers is varying in time. Often the samples can be noisy, can have varying length and both local and global structure can be important. A survey of such applications is given in [CK16].

1.1 Plan

Figure 1.1 indicates contributions in subsequent chapters: sections which contain mathematical results with ☐, those which describe algorithms with ☰ and those containing the results of computer experiments with ⊞. The remainder of this introductory chapter introduces the signature in more detail.

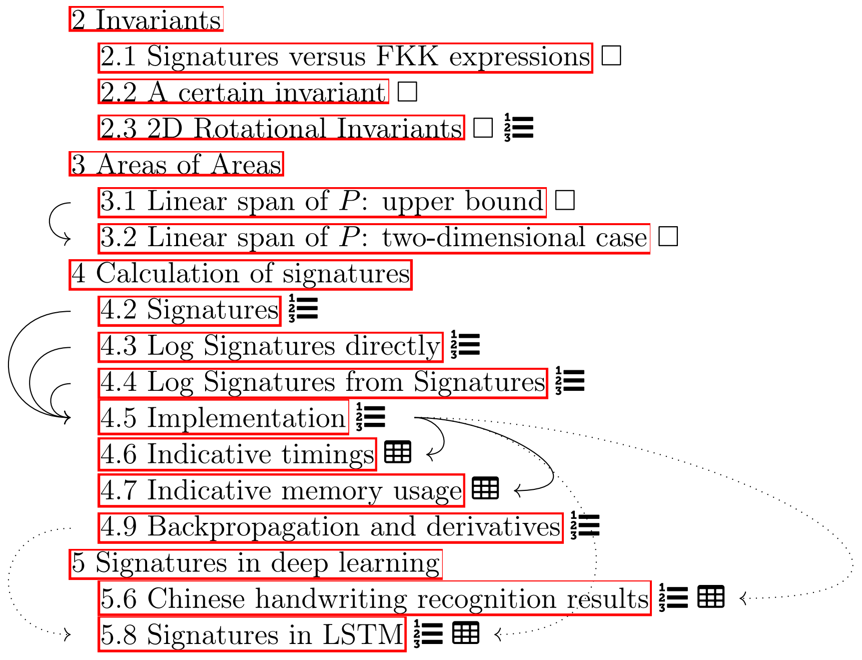

2 Invariants
2.1 Signatures versus FKK expressions ☐
2.2 A certain invariant ☐
2.3 2D Rotational Invariants ☐ ☰
3 Areas of Areas
3.1 Linear span of P: upper bound ☐
3.2 Linear span of P: two-dimensional case ☐
4 Calculation of signatures
4.2 Signatures ☰
4.3 Log Signatures directly ☰
4.4 Log Signatures from Signatures ☰
4.5 Implementation ☰
4.6 Indicative timings ⊞
4.7 Indicative memory usage ⊞
4.9 Backpropagation and derivatives ☰
5 Signatures in deep learning
5.6 Chinese handwriting recognition results ☰ ⊞
5.8 Signatures in LSTM ☰ ⊞

Figure 1.1: Plan of this document. Introductory sections to each chapter are omitted. Arrows indicate significant dependencies, with the dotted arrows indicating that although there is a logical dependency, the sections can be read independently. The fact the arrows are so few should be helpful.

1.2 What is the signature of a path?

The iterated-integral signature of a smooth path is an infinite sequence of numbers. It is used in the mathematical theory of differential equations driven by paths. In these problems, a path is the driving signal for a certain type of system. It turns out that the signature is exactly the information about a path which you need to know in order to predict how the output of the system will behave, using a generalisation of Taylor's theorem. It is natural that the signature would also be the right information to extract from a path if we want a machine-learning algorithm to understand the shape of the path.

A d-dimensional path is given by a function from an interval $[a, b] \subset \mathbb{R}$ to $\mathbb{R}^d$. We call $\mathbb{R}^d$ the *ambient space* of the curve. Its signature depends on the appearance of the path and the direction it was created, but not the speed at which it was created. If a path is modified by adding or removing a section which is exactly backtracked over, then its signature does not change [HL10]. If the path has a time dimension along which it always increases (for example it is the graph of a function of time) then exact backtracking is impossible and so any two different paths will have different signatures.[1]

The signature is divided into units called levels. We cannot store the whole signature of a path on a computer, rather we calculate a certain number of levels of it. The more levels of a signature are known, the more precisely the shape of the path is determined. If a path changes very slightly, the first few levels of its signature will also only change very slightly. If a path is moved (translated) but retains its shape, its signature will not change.

The number of elements of level m of the signature of a d-dimensional path is d^m. They are the values of iterated integrals which consist of m nested integrals, and they are labelled with m numbers each corresponding to one of the dimensions. To distinguish these numbers which label the dimensions from other numbers, we write them bold and in blue. For example, a two-dimensional path might be given in coordinates as $(\gamma_{\mathbf{1}}(t), \gamma_{\mathbf{2}}(t))$ as t varies from a to b. Its signature is a function denoted by $X^\gamma_{a,b}$ from words made of the bold blue alphabet to real numbers. Level three of its signature has eight elements, called $X^\gamma_{a,b}(\mathbf{111})$, $X^\gamma_{a,b}(\mathbf{112})$ and so on. The one indexed by the word $\mathbf{122}$ is

$$X^\gamma_{a,b}(\mathbf{122}) = \int_{t_1=a}^{b} \int_{t_2=a}^{t_1} \int_{t_3=a}^{t_2} d\gamma_{\mathbf{1}}(t_3)\, d\gamma_{\mathbf{2}}(t_2)\, d\gamma_{\mathbf{2}}(t_1). \tag{1.1}$$

[1]The signatures of two paths are the same iff they are *tree-equivalent* [HL10] which means that they only differ in terms of adding or removing pieces which consist of exact backtracking.

In general, the signature can be defined inductively on the length of the word, as an element of one level of the signature of a path is an integral involving the signature of a varying portion of the path. The signature of the empty word is the single value in level 0, and it is defined to always be 1. If w is a word and $i \in \{1, 2, \ldots, d\}$ then $X^{\gamma}_{a,b}(wi)$ is defined as the Stieltjes integral $\int_a^b X^{\gamma}_{a,t}(w)\, d\gamma_i(t)$. In the simple case that γ_i is differentiable, this is equal to $\int_a^b X^{\gamma}_{a,t}(w)\, \gamma_i'(t)\, dt$. We also use the symbol $S(\gamma)_{a,b}$ for $X^{\gamma}_{a,b}$. When the endpoints (a, b) of a path γ are clear, the signature $X^{\gamma}_{a,b}$ may be denoted $S(\gamma)$.[2]

The information in the first level of the signature is the total displacement of the path, i.e. the direction and distance from its starting point to its ending point. The information which the second level of the signature adds is the *signed area* of the path projected in each plane. Higher levels of the signature provide more detailed information about the path's shape.

1.2.1 Displacement

As mentioned, the first level of the signature of a path is just that path's total displacement vector. This is not a complicated thing, but in certain simple cases it may contain enough information about a path, or about a section of a path, to be used further. For example, it is enough information about handwritten digits to make a significant guess as to what the digit is. The Pendigits dataset [AA98] collected the traces of many people writing the digits 0 to 9. Figure 1.2 shows the displacement of 18 of each digit on a scatterplot, we see that many like digits are clustered together.

1.2.2 Signed area

For a two-dimensional path, the information carried by the first two levels of the signature is the total displacement of the path (in the first level, which is two numbers) and the signed area between the path and the straight line from its beginning to end. In higher dimensions, the second level of the signature gives the signed area of the path projected into any plane. Figure 1.3 shows this information for two straight lines in the plane and their combination, which contains area. For $d = 1$, the signature does not contain any information beyond the total displacement of the path, and is therefore not interesting. d should be considered as at least 2 in the

[2]Sometimes in analysis, the term *signature* is used in a slightly different sense: the signature of a path γ is the whole function $(c, d) \mapsto X^{\gamma}_{c,d}$, or something equivalent to it, not just that function's value on the path's endpoints. This document does not use that sense of the word.

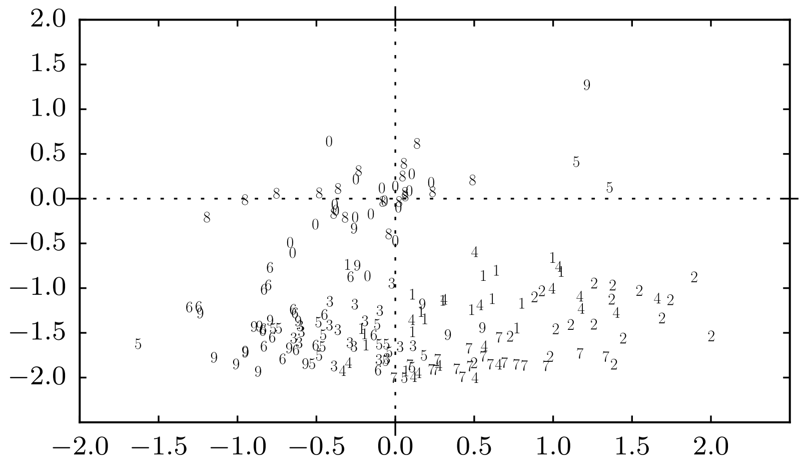


Figure 1.2: Displacement (i.e. level 1 of the signature) of the first stroke of the first 18 of each handwritten digit 0 to 9 from the training data of [AA98]. The bounding box of each digit is scaled to $[-1,1]^2$ so that the displacement lies in $[-2,2]^2$.

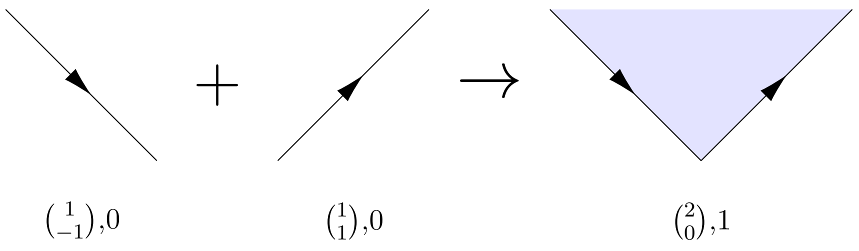


Figure 1.3: Concatenating paths and the corresponding total displacements and total signed areas.

mathematical results of this thesis.[3]

The following is an intuitive definition of the signed area of a path in the plane. For a smooth closed path, that is one which ends where it starts, the signed area is the sum of the signed areas of the regions bounded by the path, which is the area times the number of times the path goes round that region in an anticlockwise manner minus the number of times the path goes round it clockwise (i.e. the winding number). For example, in the path shown in Figure 1.4(a), regions whose areas count positively are labelled with a $+$, and negatively with a $-$. One region's area counts twice negatively; it is labelled with $--$. One enclosed region's area does not count at all, it is unlabelled. For a more general path, its signed area is the signed area of

[3]This is not to say that our methods are inapplicable to one-dimensional *data*. There are canonical ways to produce higher-dimensional paths from a path, for example adding a time dimension or a lead-lag transformation (e.g. [CK16]).

the closed path you get by joining it with a straight line from its end to its start.

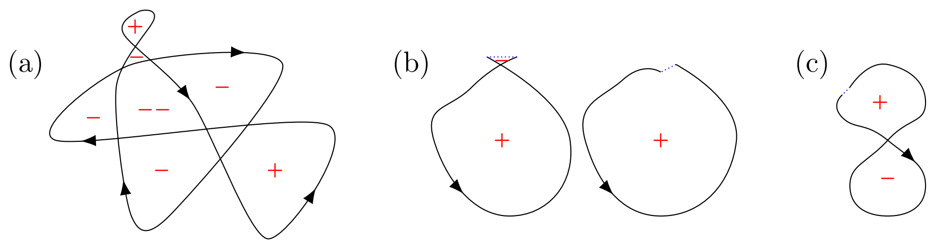


Figure 1.4: (a) A complicated closed path showing the multiplicity of each region it contains, (b) two idealised handwritten digit 0s showing the completion into a closed curve and showing how the nature of the straight line completion has only a small effect on the area, and (c) an illustration of an idealised handwritten digit 8 showing why, although it is a large object, its area might be small due to cancellation of a positive and negative part.

As an example of how the area can be useful in classifying the shape of the path, consider classifying handwritten digits 0 and 8. Usually these are written with a single stroke which ends near its beginning, so the displacement is insufficient for distinguishing them. This is reflected in Figure 1.2 which shows the 0s and 8s near the origin and intermingled. However, the signed areas are statistically different. The figure 0 is typically formed from a single anticlockwise loop, generating a positive signed area, while the figure 8 contains two regions with opposite sign, leading to cancellation of signed area. The diagrams in Figure 1.4(b) and (c) illustrate this. The histogram in Figure 1.5 shows how different the signed areas of the first (and usually only) strokes of these digits are for the same Pendigits training data. This clear separation is an illustration of the potential usefulness of the signature for machine learning.

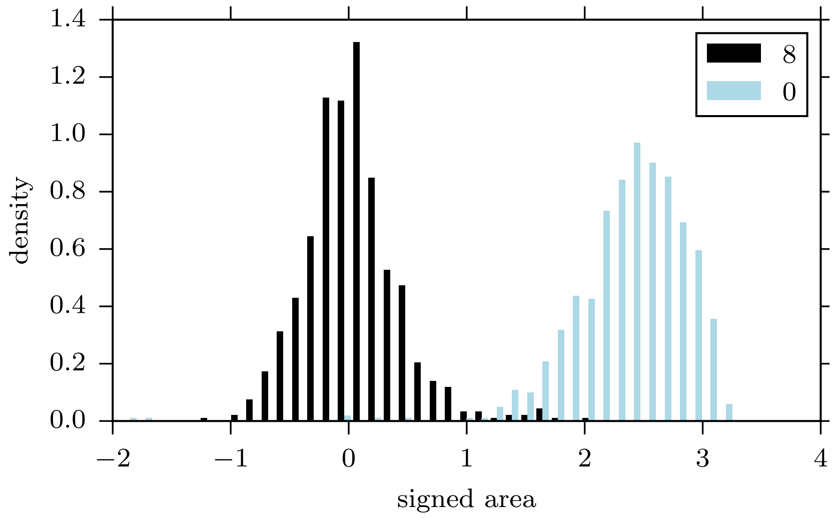


Figure 1.5: Histogram of areas of the first stroke of each 0 and 8 in the training portion of the Pendigits dataset.

I discuss later some extensions to the idea of signed area. In section 2.2 I discuss a generalisation of this concept of signed area to more than two dimensions, and in chapter 3 I discuss interpreting the whole signature, or large parts of it, as signed areas calculated recursively.

1.3 Words

In this section I introduce some background algebraic structures which I refer to repeatedly later. The main source for all this background is [Reu94]. An informal introduction is given in [Rei15]. Consider a set $\Sigma = \{\mathbf{1}, \mathbf{2}, \dots, \mathbf{d}\}$ of d "letters" which has an ordering $<$. I use these blue bold numbers as labels for the dimensions. The set of words with entries in Σ is called the *Kleene Star* of Σ and is denoted by Σ^*. The length of a word u is denoted $|u|$. The empty word is denoted by ϵ and the concatenation of words u and v is written uv. If a word w is equal to uv for some words u and v, then u is said to be a *prefix* of w and v is said to be a *suffix* of w. If u and v are both not empty then they are said to be a *proper* prefix and suffix of w. For example $\mathbf{1}$ is a proper suffix of $\mathbf{3231}$, and a suffix but not a proper suffix of $\mathbf{1}$. The ordering $<$ on Σ can be extended to an ordering $<_L$ on Σ^* called *alphabetical order* or *lexicographic order* in the usual way. (Specifically: $\epsilon <_L u$ if $|u| > 0$. For letters a and b and words u and v, $au <_L bv$ if $a < b$ or both $a = b$ and $u <_L v$.)

The *free (real) vector space* on the finite set Σ is the real vector space with basis given by the elements of Σ. We will just call it $\mathbb{R}^d$. An element looks like $a_1\mathbf{1} + \cdots + a_d\mathbf{d}$ for real numbers $a_1, \ldots, a_d$.

The *tensor algebra* of the vector space $\mathbb{R}^d$, $T(\mathbb{R}^d)$, is the set of finite sums of real multiples of words, or equivalently the set of functions from Σ^* to $\mathbb{R}$ which are zero for all but finitely many words, or equivalently the free vector space on Σ^*, $\mathbb{R}\langle\Sigma\rangle$. $T((\mathbb{R}^d))$ denotes the functions from Σ^* to $\mathbb{R}$, or formal power series on Σ considered as noncommuting, or equivalently the set of (possibly) infinite formal sums of real multiples of words, $\mathbb{R}\langle\langle\Sigma\rangle\rangle$. The word u in Σ^* is identified with the function which takes u to 1 and all other words to 0, or the expression $1u$. We are only ever interested in finite restrictions of these in order to do calculations, in particular we choose an integer m and ignore all words with length longer than m. $T^{\underline{m}}(\mathbb{R}^d)$ is the real vector space with basis given by words of length m or less[4]. The concatenation of words is extended linearly to form bilinear[5] associative operations on $T((\mathbb{R}^d))$, $T(\mathbb{R}^d)$ and $T^{\underline{m}}(\mathbb{R}^d)$ called the *concatenation product*. (On $T((\mathbb{R}^d))$ this is well defined and doesn't involve calculating infinite sums because each word is only the concatenation of finitely many pairs of words.) For example, in $T^{\underline{4}}(\mathbb{R}^3)$,[6]

$$(9\epsilon + 7\,\mathbf{132})(2\,\mathbf{1} + 4\,\mathbf{21}) = 18\,\mathbf{1} + 36\,\mathbf{21} + 14\,\mathbf{1321}.$$

Level m of the signature can be thought of as taking values in $(\mathbb{R}^d)^{\otimes m}$, which is a d^m-dimensional real vector space. In this form, the signature is seen to be an element of $T((\mathbb{R}^d))$. If $a \in T(\mathbb{R}^d)$ and $b \in T((\mathbb{R}^d))$ we can form the inner product $\langle a, b\rangle = \langle b, a\rangle$ in the word basis because this is only a sum over the finitely many terms in a. In this way $T(\mathbb{R}^d)$ is a set of linear maps $T((\mathbb{R}^d)) \to \mathbb{R}$. If $a \in T(\mathbb{R}^d)$ and X is a signature then the notations $X(a)$ and $\langle a, X\rangle$ are equivalent for the value of the signature on a.[7]

If p, q and r are real numbers with $p < q < r$ and γ is a bounded variation path $[p, r] \to \mathbb{R}^d$ then the result (from [Che58]) known as **Chen's identity** states that

$$X^{\gamma}_{p,r}(i_1 i_2 \ldots i_n) = \sum_{j=0}^{n} X^{\gamma}_{p,q}(i_1 i_2 \ldots i_j) X^{\gamma}_{q,r}(i_{j+1} i_{j+2} \ldots i_n). \tag{1.2}$$

(The products indicated with ellipses can be empty, indicating the empty word, on which any signature takes the value 1.)

[4] this is known as $T^{(m)}(\mathbb{R}^d)$ in the notation of [LCL07]

[5] i.e. linear in each argument

[6] because the concatenation of $\mathbf{132}$ and $\mathbf{21}$ is ignored

[7] There is a slight confusion with the term "signature element", as it is used for both the value of a signature on an $a \in T(\mathbb{R}^d)$ and also, sometimes, specifically for the value of a signature on a word.

Restricting up to level m, this means that

$$X^{\gamma,\underline{m}}_{p,r} = X^{\gamma,\underline{m}}_{p,q} X^{\gamma,\underline{m}}_{q,r}, \tag{1.3}$$

using the concatenation product in $T^{\underline{m}}(\mathbb{R}^d)$, where $X^{\gamma,\underline{m}}_{p,r}$ means the signature of γ on $[p, r]$ up to level m.

Given two words w_1 and w_2, their shuffle product $w_1 \sqcup\!\sqcup\, w_2$ is the multiset of words which can be formed by interleaving them, including multiplicity, which we write as a polynomial on words. For example

$$\mathbf{12} \sqcup\!\sqcup\, \mathbf{3} = \mathbf{312} + \mathbf{132} + \mathbf{123}$$

This is extended linearly to a commutative and associative operation on $T(\mathbb{R}^d)$.

The signature X of a bounded variation path obeys the following relation, for any $a, b \in T(\mathbb{R}^d)$, which is a consequence of integration by parts or the product rule for differentiation. ([Ree58], see also [Reu94, Theorem 3.2])

$$X(a)X(b) = X(a \sqcup\!\sqcup\, b) \tag{1.4}$$

In particular, therefore, not every element of $T((\mathbb{R}^d))$ is the signature of a bounded variation path. For example, we see that $X(\epsilon)$ must be 1. Allowed elements are those known as *grouplike* elements.[8]

1.4 What is the log signature of a path?

The *log signature* ([LS06], [LCL07], also *logarithmic signature*) is a compressed version of the signature. It carries the same information, but in a more compact way. It is also divided into levels. Up to level m, the log signature contains fewer numbers than the signature. Any given set of values for these numbers actually gives the log signature of some path up to that level, whereas this is not the case for signatures, because there is some redundancy in the signature. For example the first two levels of the signature of a two-dimensional path consists of $2+2^2 = 6$ numbers but we saw that this information is the path's total displacement and signed area, which can be stored in three numbers, which are exactly the first two levels of the log signature. In applications, the log signature might be less susceptible to roundoff error. The log signature is defined in terms of the signature, in a way analogous to logarithms of numbers, but can be calculated via an independent algorithm.

[8]This is equivalent to the remark in section 3.5.2 of [Reu94].

The space $T((\mathbb{R}^d))$, in which the signature of a d-dimensional path lives, has a notion of logarithm ([Reu94], chapter 3), given by

$$\log(\epsilon + T) = \sum_{n\geq 1} \frac{(-1)^{n-1}T^n}{n}, \tag{1.5}$$

where T has no ϵ component, and a notion of exponential, given by

$$\exp(T) = \epsilon + \sum_{n\geq 1} \frac{T^n}{n!}. \tag{1.6}$$

In particular, these are well-defined operations which are inverses

$$\{x \in T^{\underline{m}}(\mathbb{R}^d) \mid \langle x, \epsilon\rangle = 1\} \underset{\log}{\overset{\exp}{\leftrightarrows}} \{x \in T^{\underline{m}}(\mathbb{R}^d) \mid \langle x, \epsilon\rangle = 0\}. \tag{1.7}$$

Let S be the set which consists of the signature of every path in $\mathbb{R}^d$ truncated up to some level m. S is not the whole of the vector space $T^{\underline{m}}(\mathbb{R}^d)$, although it does *span* $T^{\underline{m}}(\mathbb{R}^d)$ (see [Die13, Lemma 8]). In fact, S forms a lower-dimensional manifold. The logarithm operation maps this manifold continuously one-to-one to a linear subspace of $T(\mathbb{R}^d)$. The image of a signature under the logarithm or its representation in a basis of this subspace is called the **log signature**. The logarithms of two signatures which agree up to level m will agree up to level m, and so the phrase "log signature of a path up to level m" is unambiguous.

The subspace in which the log signature of a path in $\mathbb{R}^d$ up to level m lives is equivalent to the free m-nilpotent Lie algebra of type d, $\mathfrak{n}_{d,m}$. The log signature is a completely compressed version of the signature up to the same level – for every value in $\mathfrak{n}_{d,m}$, there is a path with that truncated log signature. It is a Lie algebra under the bracketing operation defined by $[a, b] = ab - ba$. A clear presentation of the background to this is found in [LR95].

$\mathfrak{n}_{d,m}$ is a finite dimensional real vector space, but there is no single obvious basis for it. In order to use the log signature as an efficient representation of a path, we need to choose a fixed basis. There are two commonly used bases. They are both *Hall bases*[Hal50]. A Hall basis is made up of bracketed expressions, which are expressions involving letters combined with Lie brackets like $[\mathbf{1}, [\mathbf{2}, \mathbf{1}]]$, and it is determined by an ordering of all bracketed expressions.

- The *Lyndon basis*[Shi53]. Each basis element is labelled with a Lyndon word on $\{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{d}\}$, which is a sequence which comes earlier in lexicographic order than any of its *rotations*. (For example, the rotations of **2432** are **2243**, **3224** and **4322**. **2243** and **1213** are Lyndon words but **31** and **3224** are not.)

- The standard/canonical Hall basis, which we implement in such a way as to match CoRoPa[Lyo+10] exactly. The ordering of equal-length expressions $[A, B]$ and $[C, D]$ is defined recursively: $[A, B] < [C, D]$ if either $A < C$ or $(A = C$ and $B < D)$.

In these bases, each basis element is either a letter or a single bracketed expression, whose left and right are basis elements. We always pick an order on basis elements such that shorter bracketed expressions come before longer ones, and single letters, which are the first level, are in their natural order $\mathbf{1} < \mathbf{2} < \cdots < \mathbf{d}$.

From each bracketed expression we can form a word by deleting the brackets, which we call the expression's *foliage*. For example the foliage of $[\mathbf{1}, [\mathbf{2}, \mathbf{1}]]$ is **121**. Elements of a Hall basis are labelled by their foliages, which are called *Hall words*. In the Lyndon basis the Hall words *are* the Lyndon words.

In summary, the signature of a path is a special type of element of tensor space, called a grouplike one. It is the tensor exponential of a Lie element. This element is called the *log signature* of the path. Hall bases are practical bases for free nilpotent Lie algebras, which is where truncated log signatures take their values. A Hall basis is graded, its elements in each level provide a basis for that level. Its elements in level m are labelled with words of length m called Hall words. The term *log signature* of a path is also used for the numerical expression in terms of a Hall basis. If h is a Hall word, we write P_h for the corresponding Lie element.

1.4.1 Extracting a single log signature element from the signature

Every polynomial function on signatures can be written uniquely as a linear combination of signature elements, i.e. as a linear function on the signature. In particular, any element of the log signature can be written as an expression in terms of the signature. Here I present a simple way to find this expression.

Considering everything up to level m, there is a unique linear map π_1 defined on the truncated tensor algebra which agrees on truncated grouplike elements with the logarithm function. This is explained on page 58 of [Reu94]. (This is an example of the general principle that every polynomial function from grouplike elements / signatures to the reals extends uniquely to a linear function from $T((\mathbb{R}^d))$ to the reals. This reflects how much redundancy there is in the signature.).

Denoting the non-empty words by Σ^+, π_1 of a word w is given by

$$\pi_1(w) = \sum_{k \geq 1} \frac{(-1)^{k-1}}{k} \sum_{u_1,\dots,u_k \in \Sigma^+} \langle w, u_1 \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup \cdots \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup u_k \rangle u_1 \cdots u_k. \tag{1.8}$$

Its adjoint $\pi_1^\top$, which [Reu94] introduces in section 6.2 as π_1^*, is given by the similar expression, using the duality between shuffle and concatenation.

$$\pi_1^\top(w) = \sum_{k \geq 1} \frac{(-1)^{k-1}}{k} \sum_{u_1,\dots,u_k \in \Sigma^+} \langle w, u_1 \cdots u_k \rangle u_1 \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup \cdots \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup u_k \tag{1.9}$$

This is explained in section IV of [GK08], where $\pi_1^\top$ is called π_1'. $\pi_1^\top$ is easier to calculate than π_1 because its inner sum is simply over all decompositions of w into k words without having to think about preimages of the shuffle product.

For example,

$$\pi_1^\top(\mathbf{112}) = \mathbf{112} - \frac{1}{2}(\mathbf{1} \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup \mathbf{12} + \mathbf{11} \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup \mathbf{2}) + \frac{1}{3}\mathbf{1} \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup \mathbf{1} \sqcup\mathchoice{\mkern-7mu}{\mkern-7mu}{\mkern-3.2mu}{\mkern-3.8mu}\sqcup \mathbf{2}$$

Fixing a Hall basis, there is a well known basis of each level of tensor space called the Poincaré-Birkhoff-Witt basis or PBW basis, described around page 91 of [Reu94]. Each basis element of level m is indexed by a word of length m, we use the notation P_w for the element indexed by the word w. The PBW basis has the property that when w is a Hall word then P_w is the corresponding Lie element. Thus the notation here is consistent with our use of P_w above when w is a Hall word.

On page 108 of [Reu94] is given an explicit construction of a dual basis S for the PBW basis. This is indexed by words, and if w is a word then both S_w and P_w are elements of level $|w|$ of the tensor algebra. In particular, for any words w and w' we have that $\langle S_w, P_{w'} \rangle$ is 1 if $w = w'$ and zero otherwise.[9]

Let's say we wish to find an expression for the log signature element corresponding to the basis element labelled with Hall word h. If X is the signature, then because its logarithm can be written as $\log X = \sum_{h' \text{ Hall}} l_{h'} P_{h'}$ for constants $l.$ and we are looking for l_h, our target is

$$\langle S_h, \log X \rangle = \langle S_h, \pi_1(X) \rangle = \langle \pi_1^\top(S_h), X \rangle \tag{1.10}$$

Thus $\pi_1^\top(S_h)$ is exactly the expression we need. This is not used in any of this work but is a possible alternative for obtaining the log signature whose performance could be investigated.

[9] Angle brackets are always the inner product in the word basis described in section 1.3.

1.4.2 An illustration

The log signature up to level 4 has 8 degrees of freedom. We can write it as tensors in the Lyndon basis in the following general form.

$$\begin{array}{ccccccc} aP_{1} + bP_{2} + & cP_{12} & + dP_{112} + eP_{122} & + fP_{1112} + gP_{1122} + hP_{1222} \\ \begin{pmatrix} a \\ b \end{pmatrix} & + \begin{pmatrix} 0 & c \\ -c & 0 \end{pmatrix} + & \begin{pmatrix} \begin{pmatrix} 0 & d \\ -2d & e \end{pmatrix} \\ \begin{pmatrix} d & -2e \\ e & 0 \end{pmatrix} \end{pmatrix} \quad + & \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} 0 & f \\ -3f & g \end{pmatrix} \\ \begin{pmatrix} 3f & -2g \\ 0 & h \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} -f & 0 \\ 2g & -3h \end{pmatrix} \\ \begin{pmatrix} -g & 3h \\ -h & 0 \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{array} . \tag{1.11}$$

When exponentiated using (1.6), this expresses the signature truncated up to level 4 in the following form

$$\begin{aligned} &1 + \begin{pmatrix} a \\ b \end{pmatrix} + \frac{1}{2} \begin{pmatrix} a^2 & ab+2c \\ ab-2c & b^2 \end{pmatrix} + \frac{1}{6} \begin{pmatrix} \begin{pmatrix} a^3 & a^2b+6d+3ac \\ a^2b-12d & ab^2+6e+3bc \end{pmatrix} \\ \begin{pmatrix} a^2b+6d-3ac & ab^2-12e \\ ab^2+6e-3bc & b^3 \end{pmatrix} \end{pmatrix} \\ &+ \frac{1}{24} \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a^4 & a^3b+4a^2c+12ad+24f \\ a^3b-12ad-72f & a^2b^2+4abc+12ae+12bd+24g \end{pmatrix} \\ \begin{pmatrix} a^3b-12ad+72f & a^2b^2+4abc-24ae-24bd+12c^2-48g \\ a^2b^2+24ae-12c^2 & ab^3+4b^2c+12be+24h \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} a^3b-4a^2c+12ad-24f & a^2b^2+24bd-12c^2 \\ a^2b^2-4abc-24ae-24bd+12c^2+48g & ab^3-12be-72h \end{pmatrix} \\ \begin{pmatrix} a^2b^2-4abc+12ae+12bd-24g & ab^3-12be+72h \\ ab^3-4b^2c+12be-24h & b^4 \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{aligned} \tag{1.12}$$

Writing this out in terms of words would look like

$$\epsilon + a\mathbf{1} + b\mathbf{2} + \frac{1}{2}\Big[a^2\mathbf{11} + (ab+2c)\mathbf{12} + (ab-2c)\mathbf{21} + b^2\mathbf{22}\Big] + \cdots \tag{1.13}$$

1.5 Tools

Here I describe a number of new software tools for examining the signature which I have found useful, and which should help people in different ways trying to get to know the signature and its behaviour. They are not all original ideas.

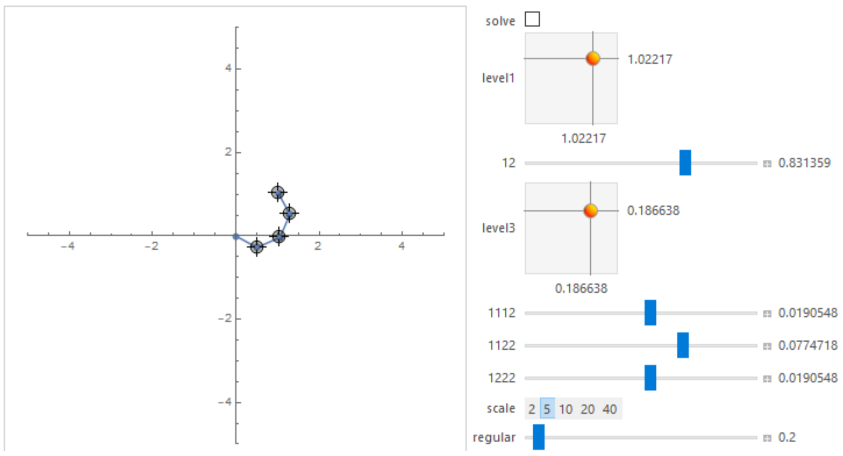
1.5.1 Five points in two dimensions

In the specific case $d = 2$, $m = 4$, the log signature has eight components, and a two dimensional path with four segments has eight degrees of freedom. The file `view.m` provides an interactive Mathematica 10 visualisation of the relationship between a path made of four segments and its log signature. It depends on another file `bch.m`, which has been generated by additional functionality in the original python logsignature code [Rei15], which defines a single function returning the log signature of a path defined by four displacements. The visualisation should appear when `view.m` is run. The log signature appears as widgets on the right of the graph of the path. Because level 1 and level 3 of the log signature are two dimensional, they are represented by 2d controls. The other log signature components are controlled separately. When 'solve' is ticked, you can gently move the widgets to change the components of the log signature and see the path move. Note that the **12** component controls the signed area enclosed by the path. The calculation of solutions is not perfect, but is enough to get a general picture. It is relying on built-in nonlinear optimisation routines in Mathematica to attempt to invert the signature in this special case. When 'solve' is unticked, you can drag the locators to change the path, and see the corresponding log signature elements move. For example, after drawing 3 paths the window might look as shown in Figure 1.6.

1.5.2 Freehand drawing

The simple tool `freehand_draw.py` lets you draw paths with your mouse in a blank window. The signature of each path is printed on the console. This is one tool I wanted to have in order to get a feel for the signature elements. Signed area and its robustness on paths which are nearly closed is particularly easy to illustrate with the tool.

For example, after drawing 3 paths the window might look as shown in Figure 1.7. The following would be printed on the console, indicating truncated log signatures up to level 3 of each path. In particular, the three lines in order correspond to (1) the letters of the word 'hey', (2) the vertical stroke of the exclamation mark and (3) the dot.



solve ☐

level1 1.02217 1.02217

12 0.831359

level3 0.186638 0.186638

1112 0.0190548

1122 0.0774718

1222 0.0190548

scale 2 [✓ 5] 10 20 40

regular 0.2

Figure 1.6: The `view.m` tool showing a path among five points. The widgets on the right show its log signature up to level 8.

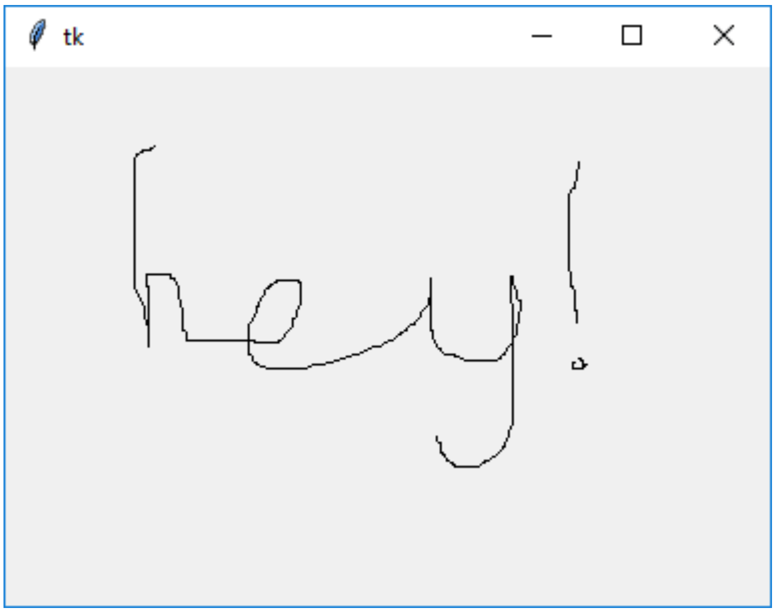


Figure 1.7: Appearance of `freehand_draw.py` after some mouse drags.

```
[ 0.366492  -0.379581   0.020182  -0.007912  -0.003532]
[-0.002618  -0.212042   0.001631  -0.000008  -0.000014]
[ 0.002618   0.005236   0.000154   0.000001   0.000001]
```

1.5.3 Arbitrary precision signature calculations

The `iisignature` project sources include a project `arbprec` which calculates signatures of paths in arbitrary precision and compares them to those using floating point arithmetic. Very often we see that these results do not differ by much for random paths. There are cases where calculating the signature of a path naturally results in a loss of floating point accuracy, for example the 1-dimensional path $(0), (a), (b)$ where $a \approx 1$ and $b \approx 0$, and so testing with arbitrary precision may be useful. Long paths given by lots of points could create a unique problem, with this type of cancellation happening in some dimensions.

1.5.4 Free lie algebra calculations

The file `free_lie_algebra.py`[10] provides objects to do many of the calculations described in [Reu94]. The main types are summarised in Table 1.1. It is useful for getting intuition and testing out conjectures, and is generally hackable.

As an example, we could try to find the linear combination of signature elements corresponding to the coefficient of **112** in the log signature in the Lyndon basis. We might do that like this using the method of subsection 1.4.1.

```
    from free_lie_algebra import *
    d=2
    m=3
    H=HallBasis(d,m,lessExpressionLyndon)

    with UseRationalContext():
        answer=pi1adjoint(S("112",H))
```

This sets `answer` to the `Elt` representing $\pi_1^\top(S_{\mathbf{112}})$ which is what we are looking for. If we run `print(answer.pretty())` we will be told

```
                [1/6]112-[1/3]121+[1/6]211
```

[10] github.com/bottler/free-lie-algebra-py

name	set
`Word`	Σ^*
`Elt`	tensor space, $K\langle\Sigma\rangle$
`EltElt`	$K\langle\Sigma\rangle^{\otimes n}$, any n
`tuple(int...)`	free magma, $M\langle\Sigma\rangle$
`HallBasis`	a Hall basis for a given dimension and depth/level.
`TensorSpaceBasis`	a basis of $K\langle\Sigma\rangle$, e.g. the PBW basis or its dual, up to a given dimension and depth/level.

Table 1.1: Types in `free_lie_algebra.h`

which fits with what we would come up with by staring at (1.12) to find a (in fact, the *unique*) linear combination of elements which adds up to d, namely that

$$d = \frac{1}{6}\Big(\frac{a^2b+6d+3ac}{6}\Big) - \frac{1}{3}\Big(\frac{a^2b-12d}{6}\Big) + \frac{1}{6}\Big(\frac{a^2b+6d-3ac}{6}\Big).$$

The library uses floating point numbers as coefficients by default. The use of `UseRationalContext` makes it use rational numbers from `sympy` [Meu+17], which is prettier in a case like this.

We can further verify this in the code. The `arbitraryGrouplikeEltSympy` function returns an arbitrary truncated signature expression as an `Elt` in the style of (1.12). The coefficients are `sympy` expressions. It is a tensor where each element in terms of its corresponding log signature elements for a given Hall basis. Instead of labelling the log signature elements with letters of the alphabet like the $a, b, c, \dots$ of (1.11), the element corresponding to a word is given a name `x` subscripted with the number which looks like the word. For example c is `x_12`. We can perform the verification like this

```
sig = arbitraryGrouplikeEltSympy(H)
print(dotprod(sig,answer).expand())
```

which prints `x_112` as desired.

1.5.5 Mathematica signature tools

Mathematica has a highly regarded system for symbolic integration. If someone has a curve given in mathematical form and wants to try to calculate its signature in closed form or numerically, it would make sense to use Mathematica. Therefore I created tools to make this easy.

`parametricCurves.wl` provides functions for manipulating parametric curves (which are represented either as a list of functions or a list of expressions). Some simple manipulations are provided (shifting, concatenating, plotting, adding a time dimension) which may be useful for experiments. The calculation of the signature, with iterative calls to Mathematica's integrator, is the main capability. For example, a calculation of the signature of one revolution of a circle up to level 3 could be requested with

```
SigFns[{Cos, Sin}, 2 Pi, 3]
```

which results in the following.

```
{{0, 0},
 {{0, Pi}, {-Pi, 0}},
 {{{0, -Pi}, {2*Pi, 0}},{{-Pi, 0}, {0, 0}}}}
```

`bases.wl` contains basic Hall basis calculations sufficient to map an expanded log signature to a basis. This is expressed in a way close enough to the mathematical presentation to be instructive. `calcSignature.wl` calculates signatures and log signatures of piecewise linear curves using straight Mathematica code. This is like the basic functionality of `iisignature`. This is again instructive rather than intended to be used for high volume calculation. I similarly provided examples of the basic calculation of the signature of a path through given points in `R` and `matlab` because people ask.

1.6 A conjecture about figures of eight

One way I attempted to get a feel for signature elements is to think about paths which are simple and whose signatures are zero up to a certain level. This is something that has been considered for example in [GK14]. In section 7.5.2 of [FV10] it is shown that among paths whose signature is a given truncated grouplike element, equivalently among all the paths with a given truncated log signature, there is a unique shortest path and it can be parameterised at constant speed. Finding it for a given truncated signature, or indeed any path, is an open problem in general. For level 1, this path is a straight line. For level 2 given, it is quite easy to find the path, it will be a piece of circle or helix. This follows from the work in [BD93]. There is no easy way to find the path for higher levels, although this would be quite informative. For level 3, a numerical attempt has recently been announced in [PSS18]. Having paths which are zero up to each level could be useful not only for gaining intuition but also because they could be combined to produce paths which agree with a given

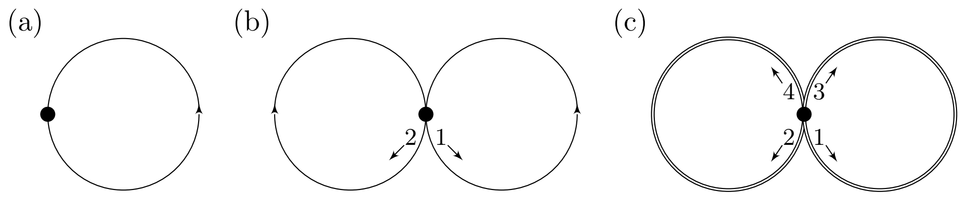


Figure 1.8: Closed paths are shown with their start and end marked with a solid circle. (a) A circle, a path whose first nonzero signature entry is on level 2. (b) A figure of eight made of two circles, a path whose first nonzero signature entry is on level 3. (c) A path made of two traverses of the same figure of eight, whose first nonzero signature entry is on level 5.

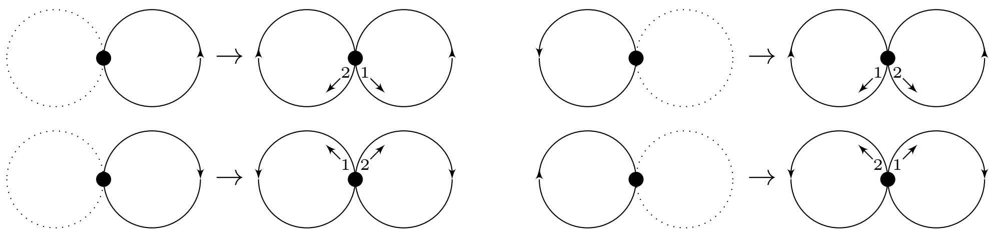


Figure 1.9: The transformation of each loop in the path to a figure of 8 which leads from each path to the next.

signature. In this section, I present an interesting pattern which looks like a supply of paths whose first non-zero signature level is arbitrarily high.

In electromagnetism, it is common to consider a dipole moment, the long-range effect of two nearby particles of opposite charge. Magnetic poles come in pairs in nature, and we often consider combinations of them in such a way that the total dipole moment is zero, leading us to quadrupole moments. The process continues. The fact that a figure of eight has nonzero values in level 3 of its signature but zero values in level 1 and 2, which is seen because it is closed and the areas of each loop (which have signed area but no displacement) come with opposite signs, is reminiscent of this process.

By numerical experiment, I have found a pattern where traversing a figure of eight several times (all starting from the central crossing point) I can get paths whose signatures are zero up to many levels. There are four ways to traverse a given figure of eight with zero area – just by choosing which direction to start.

- Two such figure-of-eight traverses can be concatenated to a path which makes a total of four loops whose first nonzero signature elements are at level 5. These paths are shown in Figure 1.8.
- Two such four-loop traverses can be concatenated (in exactly one way) to make

a path which has 8 loops and whose first nonzero signature elements are at level 8.

- Two such 8 loop traverses can be concatenated (in exactly one way) to make a path which has 16 loops and whose first nonzero signature elements are at level 13.
- Two such traverses can be concatenated in exactly one way to make a path whose signature is zero up to at least level 18.

The general process which goes from one of these paths to the next is that of replacing each single loop by a figure of eight in the pattern shown in Figure 1.9. This pattern is reminiscent of the way the Thue-Morse sequence is generated. For example, the pattern of whether each loop is clockwise or anticlockwise in one of these paths corresponds to the Thue-Morse sequence. Calculating signatures for these paths is illustrated in the file `figure8Fibonacci.py` which uses `iisignature` to calculate signatures. It is simple to conjecture that paths can be formed in this way whose first nonzero signature elements are any given Fibonacci number.

Chapter 2

Invariants

In this chapter I discuss some new results about signature elements which are invariant under certain transformations of the ambient space. Invariants to transformations which are known to be irrelevant for the problem at hand are potentially useful in machine learning as explained in subsection 5.3.4.

2.1 Signatures versus FKK expressions

There is a vast literature on invariants of curves, mostly in two dimensions. Among the techniques used, the method of "integral invariants" of [FKK10], which has been used for example in [GMW10] for character recognition, is close to our iterated-integral signature method. Their method has not been explicitly compared with our iterated-integral signature. In that work, for a curve $X : [0,T] \to \mathbb{R}^d$, $d = 2, 3$, the building blocks for invariants are expressions of the form

$$\int_0^T (X_r^1)^{\alpha_1} \dots (X_r^d)^{\alpha_d} dX_r^i, \qquad i = 1, .., d. \tag{2.1}$$

for nonnegative integers $\alpha_1 \dots \alpha_d$. This can be written using the shuffle identity (1.4) in terms of the signature $S(X)_{0,T}$ as follows.[1]

$$\begin{aligned}
&\int_0^T (X_r^1)^{\alpha_1} \dots (X_r^d)^{\alpha_d} dX_r^i \\
&\quad = \int_0^T \langle S(X)_{0,r}, \mathbf{1}^{\sqcup\!\sqcup \alpha_1} \rangle \dots \langle S(X)_{0,r}, \mathbf{d}^{\sqcup\!\sqcup \alpha_d} \rangle \, dX_r^i \\
&\quad = \int_0^T \langle S(X)_{0,r}, \mathbf{1}^{\sqcup\!\sqcup \alpha_1} \sqcup\!\sqcup \dots \sqcup\!\sqcup \mathbf{d}^{\sqcup\!\sqcup \alpha_d} \rangle \, dX_r^i \\
&\quad = \langle S(X)_{0,T}, (\mathbf{1}^{\sqcup\!\sqcup \alpha_1} \sqcup\!\sqcup \dots \sqcup\!\sqcup \mathbf{d}^{\sqcup\!\sqcup \alpha_d}) i \rangle && (2.2) \\
&\quad = \alpha_1! \dots \alpha_d! \langle S(X)_{0,T}, (\mathbf{1}^{\alpha_1} \sqcup\!\sqcup \dots \sqcup\!\sqcup \mathbf{d}^{\alpha_d}) i \rangle && (2.3)
\end{aligned}$$

These building blocks, then, are the signature elements which are a shuffle of letters concatenated with a single letter. These are exactly the signature elements which show up when defining an integral of a one form along a path. We note that the building blocks are *not* enough to uniquely characterize a path, unlike iterated-integral signatures. Indeed, the following lemma gives a counterexample to the conjecture on p.906 in [FKK10] that "signatures of non-equivalent curves are different" (here, the "signature" of a curve means the set of expressions of the form (2.1)). The idea for the counterexample is that the the whole of the first two levels of the signature,

[1] I write $a^{\sqcup\!\sqcup n}$ for $\overbrace{a \sqcup\!\sqcup \dots \sqcup\!\sqcup a}^{n \text{ times}}$, which is well-defined because the shuffle product is associative.

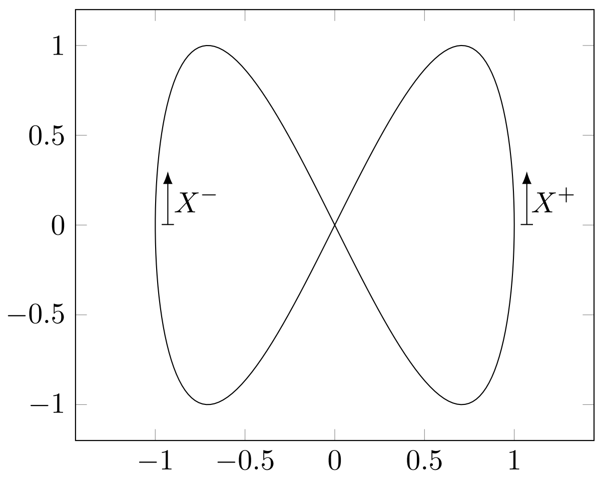


Figure 2.1: The lemniscate of Gerono. Traversing it once from each of the two starting points indicated gives two distinct closed curves with distinct iterated-integral signatures, but which cannot be distinguished with the “signature” of [FKK10].

but not the third, is included in the form (2.1), so a good choice would be something like a figure of eight, which has nothing on the first two levels, and it makes sense to pick a curve which has a tractable form.

Lemma 1. *Consider the two closed curves X^+ and X^- in $\mathbb{R}^2$, given for t in $[0, 2\pi]$ as*

$$\begin{aligned} X_t^{\pm,1} &= \pm \cos t \\ X_t^{\pm,2} &= \sin 2t. \end{aligned}$$

Then all the expressions (2.1) coincide on X^+ and X^-.

These curves both trace a figure called the *lemniscate of Gerono* which is illustrated in Figure 2.1, but in different ways.

Proof. Consider the function $f_n^m(t) := \cos^m t \, \sin^n t$, where m and n are nonnegative integers. If n is odd, then $f_n^m(t) = -f_n^m(2\pi - t)$ so $\int_0^{2\pi} f_n^m(t)\,dt$ is zero. If m is odd, then

$$\int_0^{2\pi} f_n^m(t)\,dt = -\int_{\frac{\pi}{2}}^{-\frac{3\pi}{2}} f_n^m(\frac{\pi}{2} - t)\,dt = \int_{-\frac{3\pi}{2}}^{\frac{\pi}{2}} f_m^n(t)\,dt = \int_0^{2\pi} f_m^n(t)\,dt = 0.$$

Thus $\int_0^{2\pi} f_n^m(t)\,dt$ can only be nonzero if m and n are both even.

Any expression like (2.1) is either of the form

$$\begin{aligned}
A^{\pm}_{m,n} &= \int_0^{2\pi} (X^{\pm,1}_t)^m (X^{\pm,2}_t)^n \, dX^{\pm,1}_t \\
&= \int_0^{2\pi} (\pm 1)^m \cos^m t \, \sin^n 2t \, (\mp \sin t) \, dt \\
&= \mp 2^n (\pm 1)^m \int_0^{2\pi} \cos^{m+n} t \sin^{n+1} t \, dt \\
&= \begin{cases} 0 & n \text{ even or } m \text{ even} \\ -2^n \int_0^{2\pi} \cos^{m+n} t \sin^{n+1} t \, dt & \text{otherwise} \end{cases}
\end{aligned}$$

or of the form

$$\begin{aligned}
B^{\pm}_{m,n} &= \int_0^{2\pi} (X^{\pm,1}_t)^m (X^{\pm,2}_t)^n \, dX^{\pm,2}_t \\
&= \int_0^{2\pi} (\pm 1)^m \cos^m t \, \sin^n 2t \, (2 \cos 2t) \, dt \\
&= 2^{n+1} (\pm 1)^m \int_0^{2\pi} \cos^{m+n} t \, \sin^n t \, (\cos^2 t - \sin^2 t) \, dt \\
&= \begin{cases} 0 & n \text{ odd or } m \text{ odd} \\ 2^{n+1} \int_0^{2\pi} \cos^{m+n} t \, \sin^n t \, (\cos^2 t - \sin^2 t) \, dt & \text{otherwise} \end{cases} .
\end{aligned}$$

Both these expressions are free from the symbols $\pm$ and $\mp$. Therefore these two curves have the same values on terms of the form (2.1). [2] $\square$

2.2 A certain invariant

In [DR18] we report several facts relating to signature elements of a path γ from $[0, T]$ to $\mathbb{R}^d$ which are invariant under certain groups of transformations. In this section, we report interesting results on the geometric interpretation of a certain invariant, which generalises the concepts of signed area and winding number to curves in higher dimensions.

Let

$$\mathrm{GL}(\mathbb{R}^d) = \{A \in \mathbb{R}^{d \times d} : \det(A) \neq 0\},$$

be the general linear group of $\mathbb{R}^d$.

[2] Note that X^+ and X^- are not tree-equivalent and therefore have different (iterated-integral) signatures. The lowest level on which they differ is level 4.

Definition 2. *For a positive number w, we call $\phi \in T(\mathbb{R}^d)$ a* GL ***invariant of weight*** *w if*

$$\left\langle X_{0,T}^{A\circ\gamma}, \phi \right\rangle = (\det A)^w \left\langle X_{0,T}^{\gamma}, \phi \right\rangle$$

for all $A \in \mathrm{GL}(\mathbb{R}^d)$ and all bounded variation paths $\gamma : [0,T] \to \mathbb{R}^d$.

It turns out that such invariants only exist for integer w, and that invariants of weight w live in level $m = wd$ of the signature. In this section, I label the alphabet of dimensions $\{\mathbf{1}, \ldots \mathbf{d}\}$ as $\{x_1, \ldots, x_d\}$. Whatever the dimension d of the curve's ambient space, the space of invariants of weight 1 has dimension 1 and is spanned by

$$\mathrm{Inv}_d := \mathrm{Inv}_d(x_1, .., x_d) := \sum_{\sigma \in S_d} \mathrm{sign}(\sigma)\ x_{\sigma(1)} .. x_{\sigma(d)} = \det \begin{pmatrix} x_1 & .. & x_d \\ .. & .. & .. \\ x_1 & .. & x_d \end{pmatrix}. \tag{2.4}$$

Here, for a matrix C of non-commuting variables, (compare [FW86, Definition 3.1])

$$\det C := \sum_\tau \mathrm{sign}\,\tau \prod_i C_{i\tau(i)}.$$

It is the fact that the multiplication is not commutative which makes the determinant in (2.4) not trivially zero.

This invariant is of homogeneity d, meaning it is an element of level d of tensor space, and is the subject of this section. It is well-known that the invariant for $d = 2$, Inv_2 is double the signed area of a curve, as discussed in subsection 1.2.2. The invariant for $d = 3$, Inv_3 is identified in [FKK10] who call it $-J_1$.[3] In section 3.4 of [FKK10] they interpret this invariant as an extension of the concept of signed area and as the volume of a solid. Our line of research here began with trying to give a more precise interpretation of this invariant, and with the observation from numerical experiment that for many 3-dimensional curves with only a simple bend, Inv_3 seems up to sign to coincide with six times the volume of the convex hull of the curve.

The following lemma tells us that we can write Inv_d in terms of expressions on lower levels. To state it, we first define the operation $\mathsf{InsertAfter}(x_i, r)$ on monomials of order $n \geq r$, as the insertion of the variable x_i after position r, and extend it

[3]Using equations (18) through (21) of [FKK10] we have, in their notation, $J_1 = XYZ - 2XZ^{[0,1,0]} + 2YZ^{[1,0,0]} - 2ZY^{[1,0,0]}$. In our notation, this expression as a signature element is $\mathbf{1}\sqcup\!\sqcup\mathbf{2}\sqcup\!\sqcup\mathbf{3} - 2\mathbf{1}\sqcup\!\sqcup\mathbf{23} + 2\mathbf{2}\sqcup\!\sqcup\mathbf{13} - 2\mathbf{3}\sqcup\!\sqcup\mathbf{12}$, which expands to $-\mathbf{123} - \mathbf{231} - \mathbf{312} + \mathbf{132} + \mathbf{213} + \mathbf{321} = -\mathrm{Inv}_3$.

linearly. For example

$$\begin{aligned}\mathsf{InsertAfter}(x_1,1)\,\mathrm{Inv}_2(x_2,x_3) &= \mathsf{InsertAfter}(x_1,1)\Big(x_2x_3 - x_3x_2\Big)\\ &= x_2x_1x_3 - x_3x_1x_2.\end{aligned}$$

Lemma 3. *In any dimension d and for any $r = 0, 1, \ldots, d-1$*

$$\mathrm{Inv}_d(x_1,\ldots,x_d) = (-1)^r \sum_{j=1}^{d} (-1)^{j+1} \mathsf{InsertAfter}(x_j, r)\,\mathrm{Inv}_{d-1}(x_1,\ldots,\widehat{x_j},\ldots,x_d),$$

where $\widehat{x_j}$ denotes the omission of that argument.

For d odd, this simplifies to

$$\mathrm{Inv}_d(x_1,\ldots,x_d) = \sum_{j=1}^{d} (-1)^{j+1} x_j \sqcup\!\sqcup\, \mathrm{Inv}_{d-1}(x_1,\ldots,\widehat{x_j},\ldots,x_d).$$

Proof. The first statement follows from expressing the determinant in (2.4) in terms of minors with respect to the row $r+1$ (since the x_i are non-commuting, this does not work with columns!).

Regarding the second statement, since d is odd and then using the first statement

$$\begin{aligned}\mathrm{Inv}_d &= \sum_{r=0}^{d-1} (-1)^r\, \mathrm{Inv}_d\\ &= \sum_{r=0}^{d-1} (-1)^r (-1)^r \sum_{j=1}^{d} (-1)^{j+1} \mathsf{InsertAfter}(x_j, r)\,\mathrm{Inv}_{d-1}(x_1,..,\widehat{x_j}..,x_d)\\ &= \sum_{j=1}^{d} (-1)^{j+1} \sum_{r=0}^{d-1} \mathsf{InsertAfter}(x_j, r)\,\mathrm{Inv}_{d-1}(x_1,..,\widehat{x_j}..,x_d)\\ &= \sum_{j=1}^{d} (-1)^{j+1} x_j \sqcup\!\sqcup\, \mathrm{Inv}_{d-1}(x_1,..,\widehat{x_j}..,x_d),\end{aligned}$$

as claimed. □

An immediate consequence is the following lemma, where we are considering a curve $X : [0,T] \to \mathbb{R}^d$ and its signature $S(X) = S(X)_{0,T}$.

Lemma 4. *If the ambient dimension d is odd and the curve X is closed (i.e. $X_T =$*

X_0) then

$$\left\langle S(X)_{0,T}, \mathrm{Inv}_d \right\rangle = 0.$$

Proof. By Lemma 3 and then by the shuffle identity (1.4)

$$\begin{aligned}\left\langle S(X)_{0,T}, \mathrm{Inv}_d \right\rangle &= \sum_{j=1}^{d} \left\langle S(X)_{0,T}, (-1)^{j+1} x_j \sqcup\!\sqcup \mathrm{Inv}_{d-1}(x_1, .., \widehat{x_j} .., x_d) \right\rangle \\ &= \sum_{j=1}^{d} (-1)^{j+1} \left\langle S(X)_{0,T}, x_j \right\rangle \left\langle S(X)_{0,T}, \mathrm{Inv}_{d-1}(x_1, .., \widehat{x_j} .., x_d) \right\rangle \\ &= 0,\end{aligned}$$

since the increment $\left\langle S(X)_{0,T}, x_j \right\rangle = X^j_T - X^j_0$ is zero for all j by assumption. □

In even dimension we have the phenomenon that closing a curve does not change the value of the invariant. We mentioned that this was true for Inv_2 in subsection 1.2.2.

Lemma 5. *If the ambient dimension d is even, then for any curve X*

$$\left\langle S(X), \mathrm{Inv}_d \right\rangle = \left\langle S(\bar{X}), \mathrm{Inv}_d \right\rangle,$$

where $\bar{X}$ is X concatenated with the straight line connecting X_T to X_0.

Proof. Let $\bar{X}$ be parametrized on $[0, 2T]$ as follows: $\bar{X} = X$ on $[0, T]$ and it is the linear path connecting X_T to X_0 on $[T, 2T]$. By translation invariance we can assume $X_0 = 0$ and by $\mathrm{GL}(\mathbb{R}^d)$-invariance that X_T lies on the x_1 axis. Then the only component of $\bar{X}$ that is non-constant on $[T, 2T]$ is the first one, $\bar{X}^1$.

By Lemma 3

$$\mathrm{Inv}_d = -\sum_{j=1}^{d} (-1)^{j+1} \mathrm{Inv}_{d-1}(x_1, \ldots, \hat{x}_j, \ldots, x_d) x_j.$$

Letting the summands act on $S(\bar{X})_{0,t}$ we get ± 1 times

$$\int_0^t \left\langle S(\bar{X})_{0,r}, \mathrm{Inv}_{d-1}(x_1, \ldots, x_d) \right\rangle d\bar{X}^j_r.$$

For $j \neq 1$ these expressions are constant on $[T, 2T]$, since we arranged things so that those $\bar{X}^j$ do not move on $[T, 2T]$. But also for $j = 1$ this expression is constant on

$[T, 2T]$. Indeed, the integrand

$$\Big\langle S(\bar{X})_{0,r}, \mathrm{Inv}_{d-1}(x_2, x_3, \ldots, x_d)\Big\rangle,$$

is zero on $[T, 2T]$, since X, projected on the $x_2 - \cdots - x_d$ hyperplane, is a closed curve, and so Lemma 4 applies. □

Lemma 6. *Let X be the piecewise linear curve through $p_0, .., p_d \in \mathbb{R}^d$. Then*

$$\Big\langle S(X)_{0,T}, \mathrm{Inv}_d\Big\rangle = \det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_0 & p_1 & .. & p_d \end{bmatrix}$$

Proof. First, for any $v \in \mathbb{R}^d$,

$$\det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_0 + v & p_1 + v & .. & p_d + v \end{bmatrix} = \det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_0 & p_1 & .. & p_d \end{bmatrix}.$$

Since the signature is also invariant to translation, we can therefore assume $p_0 = 0$. Now both sides of the statement transform the same way under the action of $\mathrm{GL}(\mathbb{R}^d)$ on the points $p_1, ..p_d$. It is then enough to prove this for

$$\begin{aligned} p_0 &= 0 \\ p_1 &= e_1 \\ p_2 &= e_1 + e_2 \\ &.. \\ p_d &= e_1 + .. + e_d. \end{aligned}$$

Now, for this particular choice of points the right hand side is clearly equal to 1. For the left hand side, the only non-zero term is

$$\begin{aligned} \Big\langle S(X)_{0,T}, \mathbf{12..d}\Big\rangle &= \int dX^1..dX^d \\ &= 1. \end{aligned}$$

□

The modulus of the determinant

$$\det \begin{bmatrix} 1 & 1 & .. & 1 \\ 0 & p_1 & .. & p_d \end{bmatrix} = \det \begin{bmatrix} p_1 & .. & p_d \end{bmatrix}$$

gives the Lebesgue measure of the parallelepiped which is spanned by the vectors

$p_1 - p_0, .., p_d - p_0$. The polytope spanned by the points $p_0, p_1, .., p_d$ fits $d!$ times into that parallelepiped. We hence have the relation to classical volume as follows.

Lemma 7. *Let $p_0, .., p_d \in \mathbb{R}^d$, then*

$$|\operatorname{Convex-Hull}(p_0, .., p_d)| = \frac{1}{d!} \left| \det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_0 & p_1 & .. & p_d \end{bmatrix} \right|$$

We now proceed to piecewise linear curves with more than d vertices.

Lemma 8. *Let X be the piecewise linear curve through, $p_0, .., p_n \in \mathbb{R}^d$, with $n \geq d$. Then, for certain choices of i,*

$$\left\langle S(X)_{0,T}, \operatorname{Inv}_d \right\rangle = \sum_i \det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_{i_0} & p_{i_1} & .. & p_{i_d} \end{bmatrix}. \tag{2.5}$$

For d even, *the subsequences i are chosen as follows:*

$$i_0 = 0$$

and $i_1, .., i_d$ ranges over all possible increasing subsequences of $1, 2, .., n$ such that for ℓ odd: $i_\ell + 1 = i_{\ell+1}$.

For d odd, *they are chosen as follows:*

$$i_0 = 0$$
$$i_d = n,$$

and $i_1, .., i_{d-1}$ ranges over all possible increasing subsequences of $1, 2, .., n-1$ such that for ℓ odd: $i_\ell + 1 = i_{\ell+1}$.

Remark 9. *The number of indices is easily calculated. In the even case, we have $B := d/2$ "groups of two" to place, $A := n - d$ "fillers" in between. This gives*

$$\binom{A+B}{B} = \binom{n - d + d/2}{d/2} = \binom{\lfloor \frac{d}{2} \rfloor + n - d}{\lfloor \frac{d}{2} \rfloor},$$

where $\lfloor r \rfloor$ is the largest integer less than or equal to r.

In the odd case, we have $B := (d-1)/2$ "groups of two" to place, with $A := n - 1 - (d-1)$ "fillers" in between. This gives

$$\binom{A+B}{B} = \binom{n - 1 - \frac{d-1}{2}}{\frac{d-1}{2}} = \binom{\lfloor \frac{d}{2} \rfloor + n - d}{\lfloor \frac{d}{2} \rfloor}.$$

Example 10. *For $d = 2$, $n = 5$ we get the subsequences*

$$[0, 1, 2]$$
$$[0, 2, 3]$$
$$[0, 3, 4]$$

For $d = 4$, $n = 7$ we get the subsequences

$$[0, 1, 2, 3, 4]$$
$$[0, 1, 2, 4, 5]$$
$$[0, 1, 2, 5, 6]$$
$$[0, 2, 3, 4, 5]$$
$$[0, 2, 3, 5, 6]$$
$$[0, 3, 4, 5, 6]$$

For $d = 5$, $n = 8$ we get the subsequences

$$[0, 1, 2, 3, 4, 7]$$
$$[0, 1, 2, 4, 5, 7]$$
$$[0, 1, 2, 5, 6, 7]$$
$$[0, 2, 3, 4, 5, 7]$$
$$[0, 2, 3, 5, 6, 7]$$
$$[0, 3, 4, 5, 6, 7]$$

Proof of Lemma 8. **The case** $d = 2$
Let X be the curve through the points $p_0, p_1, .., p_n$. We can write it as concatenation of the curves $X^{(i)}$, where $X^{(i)}$ is the curve through the points p_0, p_i, p_{i+1}, p_0. The time-interval of definition for these curves (and all curves in this proof) do not matter, so we omit the subscript of $S(.)$. Then, by Chen's formula (1.3)

$$\begin{aligned}\left\langle S(X), \mathbf{12} - \mathbf{21} \right\rangle &= \left\langle S(X^{(n-1)}) \cdot .. \cdot S(X^{(1)}), \mathbf{12} - \mathbf{21} \right\rangle \\ &= \sum_{i=1}^{n-1} \left\langle S(X^{(i)}), \mathbf{12} - \mathbf{21} \right\rangle.\end{aligned}$$

For the last equality we used that

$$\Big\langle gh, \mathbf{12}-\mathbf{21}\Big\rangle = \Big\langle g, \mathbf{12}-\mathbf{21}\Big\rangle + \Big\langle h, \mathbf{12}-\mathbf{21}\Big\rangle + \Big\langle g, \mathbf{1}\Big\rangle\Big\langle h, \mathbf{2}\Big\rangle - \Big\langle g, \mathbf{2}\Big\rangle\Big\langle h, \mathbf{1}\Big\rangle,$$

and that the increments of all curves $X^{(i)}$ are zero. Now by Lemma 5 we can omit the last straight line in every $X^{(i)}$ and hence by Lemma 6

$$\Big\langle S(X^{(i)}), \mathbf{12}-\mathbf{21}\Big\rangle = \det\begin{bmatrix} 1 & 1 & 1 \\ p_0 & p_i & p_{i+1}\end{bmatrix},$$

which finishes the proof for $d = 2$.

Now assume the statement is true for all dimensions strictly smaller than some d. We show it is true for d.

***d* is odd**

As before we can assume $p_0 = 0$ and that p_n lies on the x_1 axis. Every sequence summed over on the right-hand side of (2.5) is of the form $i = (0, ..., n)$. For each of those, we calculate

$$\det\begin{bmatrix} 1 & 1 & .. & 1 & 1 \\ p_{i_0} & p_{i_1} & .. & p_{i_{d-1}} & p_{i_d}\end{bmatrix} = \det\begin{bmatrix} 1 & 1 & .. & 1 & 1 \\ 0 & p_{i_1} & .. & p_{i_{d-1}} & \Delta \cdot e_1\end{bmatrix}$$
$$= \Delta \cdot \det\begin{bmatrix} 1 & 1 & .. & 1 \\ 0 & \bar{p}_{i_1} & .. & \bar{p}_{i_{d-1}}\end{bmatrix}.$$

Here $\bar{p}_j \in \mathbb{R}^{d-1}$ is obtained by deleting the first coordinate of p_j, e_1 is the first canonical coordinate vector in $\mathbb{R}^d$ and $\Delta := (p_0 - p_n)_1 = \langle S(X), x_1\rangle$ is the total increment of X in the x_1 direction. Here we used that d is odd (otherwise we would get a prefactor -1).

The last determinant is the expression for the summands of the right-hand side of (2.5), but with dimension $d-1$ and points $0 = \bar{p}_0, \bar{p}_1, .., \bar{p}_{n-1}$. By assumption, summing up all these determinants gives

$$\Delta \cdot \Big\langle S(\bar{X}), \mathrm{Inv}_{d-1}\Big\rangle = \Big\langle S(X), x_1\Big\rangle\Big\langle S(\bar{X}), \mathrm{Inv}_{d-1}\Big\rangle,$$

where $\bar{X}$ is the curve in $\mathbb{R}^{d-1}$ through the points $\bar{p}_0, ..\bar{p}_{n-1}$. Since $\bar{p}_n = \bar{p}_0 = 0$, we can attach the additional point $\bar{p}_n$ to $\bar{X}$ without changing the value here (Lemma 5). Hence the sum of determinants is equal to

$$\Big\langle S(X), x_1\Big\rangle\Big\langle S(X), \mathrm{Inv}_{d-1}(x_2, .., x_d)\Big\rangle.$$

Since we arranged matters such that $\left\langle S(X), x_i \right\rangle = 0$ for $i \neq 1$, this is equal to

$$\sum_{i=1}^{d} \left\langle S(X), x_i \right\rangle \left\langle S(X), \mathrm{Inv}_{d-1}(x_1, x_2, .., \hat{x}_i, .., x_d) \right\rangle$$
$$= \left\langle S(X), \sum_{i=1}^{d} x_i ⧢ \mathrm{Inv}_{d-1}(x_1, x_2, .., \hat{x}_i, .., x_d) \right\rangle,$$

where we used the shuffle identity. By the second part of Lemma 3 this is equal to $\langle S(X), \mathrm{Inv}_d \rangle$, which finishes the proof for odd d.

***d* is even**

We proceed by induction on n. For $n = d$ the statement follows from Lemma 6.

Let it be true for some n, we show it for a piecewise linear curve through some points $p_0, .., p_{n+1}$. Write $X = X' \sqcup X''$ where X' is the linear interpolation of $p_0, .., p_n$, X'' is the linear path from p_n to p_{n+1}, where $\sqcup$ denotes concatenation of paths. By assumption, (2.5) is true for the curve X'. Adding an additional point p_{n+1}, the sum on the right hand side of (2.5) gets additional indices of the form

$$(p_{j_0}, .., p_{j_{d-1}}, p_{n+1}),$$

where

$$j_0 = 0$$
$$j_{d-1} = n,$$

and where $j_1, .., j_{d-2}$ ranges over all possible increasing subsequences of $1, 2, .., n-1$ such that for ℓ odd $j_\ell + 1 = j_{\ell+1}$.

Assume $p_{n+1} - p_n = \Delta \cdot e_1$ lies on the x_1-axis. Then, summing over those j,

$$\begin{aligned}
LHS &= \sum_j \det \begin{bmatrix} 1 & 1 & .. & 1 & 1 & 1 \\ 0 & p_{j_1} & .. & p_{j_{d-2}} & p_n & p_{n+1} \end{bmatrix} \\
&= \sum_j \det \begin{bmatrix} 1 & 1 & .. & 1 & 1 & 1 \\ -p_n & p_{j_1} - p_n & .. & p_{j_{d-2}} - p_n & 0 & p_{n+1} - p_n \end{bmatrix} \\
&= \sum_j \det \begin{bmatrix} 1 & 1 & .. & 1 & 1 & 1 \\ -p_n & p_{j_1} - p_n & .. & p_{j_{d-2}} - p_n & 0 & \Delta \cdot e_1 \end{bmatrix} \\
&= -\Delta \cdot \sum_j \det \begin{bmatrix} 1 & 1 & .. & 1 & 1 \\ -\bar{p}_n & \bar{p}_{j_1} - \bar{p}_n & .. & \bar{p}_{j_{d-2}} - \bar{p}_n & 0 \end{bmatrix} \\
&= -\Delta \cdot \sum_j \det \begin{bmatrix} 1 & 1 & .. & 1 & 1 \\ 0 & \bar{p}_{j_1} & .. & \bar{p}_{j_{d-2}} & \bar{p}_n \end{bmatrix} \\
&= -\Delta \cdot \Big\langle S(\bar{X}'), \mathrm{Inv}_{d-1} \Big\rangle \\
&= -\Delta \cdot \Big\langle S(X'), \mathrm{Inv}_{d-1}(x_2, .., x_d) \Big\rangle
\end{aligned}$$

Here $\bar{X}'$ is the curve in $\mathbb{R}^{d-1}$ through the points $\bar{p}_0, ..\bar{p}_n$, and we used the fact that the indices j here range over the ones used for (2.5) in dimension $d-1$ on the points $\bar{p}_0, .., \bar{p}_n$.

On the other hand, using Chen's formula, (1.3),

$$\begin{aligned}
\Big\langle S(X), \mathrm{Inv}_d \Big\rangle &= \Big\langle S(X'')S(X'), \mathrm{Inv}_d \Big\rangle \\
&= \Big\langle S(X'), \mathrm{Inv}_d \Big\rangle - \Big\langle S(X'), \mathrm{Inv}_{d-1}(x_2, .., x_d) \Big\rangle \Big\langle S(X''), x_1 \Big\rangle.
\end{aligned}$$

Here we used that $S(X'') = \exp(\Delta \cdot x_1) = 1 + \Delta \cdot x_1 + O(x_1^2)$ ([FV10, Example 7.21]), the fact that each monomial in Inv_d has exactly one occurrence of x_1 and Lemma 3. This finishes the proof. $\square$

Definition 11. *Let $X : [0,T] \to \mathbb{R}^d$ be any curve. Define its **signed volume** to be the following limit, if it exists,*

$$\text{Signed-Volume}(X) := \frac{1}{d!} \lim_{|\pi| \to 0} \sum_i \det \begin{bmatrix} 1 & 1 & .. & 1 \\ X_{t^\pi_{i_0}} & X_{t^\pi_{i_1}} & .. & X_{t^\pi_{i_d}} \end{bmatrix}.$$

Here $\pi = (0 = t^\pi_0, .., t^\pi_{n^\pi} = T)$ is a partition of the interval $[0,T]$ and $|\pi|$ denotes its mesh size. The indices i are chosen as in Lemma 8.

Theorem 12. *Let $X : [0,T] \to \mathbb{R}^d$ be a continuous curve of bounded variation.*

Then its signed volume exists and

$$\text{Signed-Volume}(X) = \frac{1}{d!}\Big\langle S(X)_{0,T}, \text{Inv}_d \Big\rangle$$

Proof. Fix some sequence $\{\pi^n\}_{n\in\mathbb{N}}$ of partitions of $[0,T]$ with $|\pi^n| \to 0$ and interpolate X linearly along each π^n to obtain a sequence of linearly interpolated curves X^n. Then by Lemma 8

$$\text{Signed-Volume}(X^n) = \frac{1}{d!}\Big\langle S(X^n)_{0,T}, \text{Inv}_d \Big\rangle$$

By stability of the signature in the class of continuous curves of bounded variation ([FV10, Proposition 1.28, Proposition 2.7]), we get convergence

$$\Big\langle S(X^n)_{0,T}, \text{Inv}_d \Big\rangle \to \Big\langle S(X)_{0,T}, \text{Inv}_d \Big\rangle$$

and this is independent of the particular sequence π^n chosen. □

The previous theorem is almost a tautology, but there are relations to classical objects in geometry. For $d = 2$, as we have seen

$$\frac{1}{2}\Big\langle S(X)_{0,T}, \text{Inv}_2 \Big\rangle,$$

is equal to the signed area of the curve X. In general dimension, the value of the invariant is related to some kind of classical "volume" if the curve satisfies some kind of monotonicity. This is in particular satisfied for the "moment curve".

Lemma 13. *Let X be the moment curve*

$$X_t = (t, t^2, ..., t^d) \in \mathbb{R}^d.$$

Then for any $T > 0$

$$\frac{1}{d!}\Big\langle S(X)_{0,T}, \text{Inv}_d \Big\rangle = |\,\text{Convex-Hull}(X_{[0,T]})|$$

Remark 14. *It is easily verified that for integers $n_1..n_d$ one has*

$$\frac{1}{n_1 \cdot .. \cdot n_d}\int_0^T dt_1^{n_1}..dt_d^{n_d} = \frac{1}{n_1}\frac{1}{n_1+n_2}..\frac{1}{n_1+..+n_d}T^{n_1+..+n_d}.$$

We deduce that

$$|\operatorname{Convex-Hull}(X_{[0,T]})| = T^{1+2+..+d} \sum_{\sigma \in S_d} \operatorname{sign} \sigma \frac{1}{\sigma(1)} \frac{1}{\sigma(1)+\sigma(2)} .. \frac{1}{\sigma(1)+..+\sigma(d)}.$$

In [KS53, Section 15], the value of this volume is determined, for $T = 1$, as

$$\prod_{\ell=1}^{d} \frac{(\ell-1)!(\ell-1)!}{(2\ell-1)!}.$$

We hence get the combinatorial identity

$$\prod_{\ell=1}^{d} \frac{(\ell-1)!(\ell-1)!}{(2\ell-1)!} = \sum_{\sigma \in S_d} \operatorname{sign} \sigma \frac{1}{\sigma(1)} \frac{1}{\sigma(1)+\sigma(2)} .. \frac{1}{\sigma(1)+..+\sigma(d)}.$$

Proof. For $n \geq d$ let $0 = t_0 < .. < t_n \leq T$ be time-points, let $p_i := X_{t_i}$ be the corresponding points on the moment curve and denote by X^n the piecewise linear curve through those points. We will show

$$\frac{1}{d!} \Big\langle S(X^n)_{0,T}, \operatorname{Inv}_d \Big\rangle = |\operatorname{Convex-Hull}(X^n_{[0,T]})|.$$

First note that for any $1 \leq i_0 < i_1 < .. \leq i_d \leq n$,

$$\det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_{i_0} & p_{i_1} & .. & p_{i_d} \end{bmatrix} = \prod_{0 \leq \ell < k \leq n} (t_{i_k} - t_{i_\ell}) > 0, \tag{2.6}$$

since it is a Vandermonde determinant.

We will decompose $P := \{p_0, .., p_n\}$ into (overlapping) sets S_ℓ with cardinality $d+1$ and such that[4]

$$|\operatorname{Convex-Hull}(p_0, .., p_n)| = \sum_{\ell} |\operatorname{Convex-Hull}(S_\ell)|.$$

A *face* of P is a subset $F \subset P$ such that its convex hull Convex-Hull(F) equals the intersection of Convex-Hull(P) with some affine hyperspace. A face is a *facet*, if its affine span has dimension $d-1$. The following is a fact that is true for any polytope spanned by some points P: up to a set of measure zero, for every point x in Convex-Hull(P), the line connecting p_0 to x exits Convex-Hull$(p_0, .., p_n)$

[4]The following can be formulated in terms of *pulling triangulations*, compare [GOT17, Chapter 16], [Lee91]. For a proof that the pulling triangulation is in fact a triangulation, see [Stu96, Proposition 8.6].

through a unique facet of Convex-Hull$(p_0, .., p_n)$ contained in $\{p_1, .., p_n\}$. Hence

$$|\operatorname{Convex-Hull}(p_0, .., p_n)| = \sum_F |\operatorname{Convex-Hull}(p_0 \cup F)|,$$

where the sum is over all such facets.

Our points p_i lie on the moment curve. Then, by (2.6), any collection of points $p_{i_0}, p_{i_1}, .., p_{i_d}$ is in general position. This means that every facet of P must have exactly d points (and not more). Facets of Convex-Hull(P) with d points are characterized by Gale's criterion ([Gal63, Theorem 3], [Zie13, Theorem 0.7]):

the points $p_{i_1}, .., p_{i_d}$, with distinct $i_j \in \{0, .., n\}$ form a facet of P if and only if any two elements of $\{0, .., n\} \setminus \{i_1, .., i_d\}$ are separated by an even number of elements in $\{i_1, .., i_d\}$.[5]

***d* odd**

We are looking for such $\{i_j\}$ such that $i_1 \geq 1$. Those are exactly the indices with

- $i_{\ell+1} = i_\ell + 1$ for ℓ odd
- $i_d = n$.

Together with $i_0 := 0$ these form the indices of Lemma 8.

***d* even**

We are looking for such $\{i_j\}$ such that $i_1 \geq 1$. Those are exactly the indices with

- $i_{\ell+1} = i_\ell + 1$ for ℓ odd.

Together with $i_0 := 0$ these form the indices of Lemma 8.

Hence

$$|\operatorname{Convex-Hull}(X^n_{[0,T]})| = \sum_i |\operatorname{Convex-Hull}(p_{i_0}, .., p_{i_d})|.$$

Now by Lemma 7

$$|\operatorname{Convex-Hull}(p_{i_0}, .., p_{i_d})| = \frac{1}{d!}\left|\det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_{i_0} & p_{i_1} & .. & p_{i_d} \end{bmatrix}\right|.$$

The determinant is in fact positive here, by (2.6). We can hence omit the modulus

[5] For example, with $n = 4$ and dimension $d = 3$, the indices $\{0, 1, 2\}$, $\{0, 2, 3\}$, $\{0, 3, 4\}$, $\{0, 1, 4\}$, $\{1, 2, 4\}$ and $\{2, 3, 4\}$ lead to the facets, which in this dimension are triangles.

and get

$$\begin{aligned}
|\,\text{Convex-Hull}(X^n_{[0,T]})| &= \sum_i |\,\text{Convex-Hull}(p_{i_0}, .., p_{i_d})| \\
&= \sum_i \frac{1}{d!} \det \begin{bmatrix} 1 & 1 & .. & 1 \\ p_{i_0} & p_{i_1} & .. & p_{i_d} \end{bmatrix} \\
&= \frac{1}{d!} \left\langle S(X^n)_{0,T}, \text{Inv}_d \right\rangle,
\end{aligned}$$

by Lemma 8.

The statement of the lemma now follows by piecewise linear approximation of X using continuity of the convex hull, which follows from [EN11, Lemma 3.2], and of iterated integrals [FV10, Proposition 1.28, Proposition 2.7]. □

2.3 2D Rotational Invariants

Note: Unlike the rest of this thesis, there are parts of this section where we consider vector spaces and tensor algebras over a field other than $\mathbb{R}$.

If ϕ is an element of $T(\mathbb{R}^2)$ or $T(\mathbb{C}^2)$ we say it is *rotationally invariant*, or *SO invariant* (SO for the *Special Orthogonal* group), if $\langle \phi, X^x_{0,t}\rangle = \langle \phi, X^{A\circ x}_{0,t}\rangle$ for all bounded variation paths $x : [0,T] \to \mathbb{R}^2$ and all 2×2 rotation matrices A.[6] A spanning set for signature elements of a two-dimensional path which are invariant under rotation is given by Theorems 2 and 3 of [Die13] as follows.

Theorem 15 (Theorem 2 of [Die13]). *Let $n \geq 2$ and $i_1, \ldots, i_n \in \{1, 2\}$ be such that*

$$\#\{k : i_k = 1\} = \#\{k : i_k = 2\}. \tag{2.7}$$

Then

$$\phi := c_{i_1 \ldots i_n} \tag{2.8}$$

is rotation invariant, where

$$\begin{aligned}
c_{i_1 \ldots i_n} &:= z_{i_1} \cdot z_{i_2} \cdot \ldots \cdot z_{i_n} \\
z_1 &:= \mathbf{1} + i\mathbf{2} \\
z_2 &:= \mathbf{1} - i\mathbf{2}.
\end{aligned}$$

[6]The signature $X^x_{0,T}$ lives in $T((\mathbb{R}^2))$ but in the complex case of this definition I am identifying it with its image under the obvious injection into $T((\mathbb{C}^2))$.

Theorem 16 (Theorem 3 of [Die13]). *Let $\phi \in T(\mathbb{R}^2)$ be rotation invariant. Then we can write ϕ as a finite sum of multiples of the invariants given in Theorem 15.*

We can tidy this up to get a basis.

Theorem 17. *The invariants given in Equation (2.8) are a basis over $\mathbb{C}$ for the space of* SO *invariants on level n in $T(\mathbb{C}^2)$.*

Proof. Let $x_1 = \mathbf{1}$ and $x_2 = \mathbf{2}$. Then the elements $x_{j_1}, \ldots, x_{j_n}$ for sequences $j_1, \ldots, j_n \in \{1, 2\}$ form a basis of level n of $T(\mathbb{C}^2)$ as a vector space over $\mathbb{C}$. The map $(x_1, x_2) \mapsto (z_1, z_2)$ is an invertible linear map. Thus $z_{j_1}, \ldots, z_{j_n}$ is also a basis of level n of $T(\mathbb{C}^2)$ as a vector space over $\mathbb{C}$. The invariants in question are a subset of this basis, so they are linearly independent. □

Theorem 18. *The real and imaginary parts of those invariants given in Equation (2.8) where $z_1 = 1$ together form a basis for the space of* SO *invariants on level n in $T(\mathbb{R}^2)$.*

Proof. By the previous theorem, the space of SO invariants on level n in $T(\mathbb{C}^2)$ is spanned freely by the set of

$$z_{j_1} \cdot .. \cdot z_{j_n} \quad \text{with} \quad \#\{r : j_r = 1\} = \#\{r : j_r = 2\}.$$

Considering sums and differences of the pairs $\{z_{j_1} \cdot .. \cdot z_{j_n}, z_{3-j_1} \cdot .. \cdot z_{3-j_n}\}$, we get that the space of SO invariants on level n in $T(\mathbb{C}^2)$ is spanned freely by the set of

$$(z_{j_1} \cdot .. \cdot z_{j_n} + z_{3-j_1} \cdot .. \cdot z_{3-j_n}) \quad \text{and} \quad (z_{j_1} \cdot .. \cdot z_{j_n} - z_{3-j_1} \cdot .. \cdot z_{3-j_n})$$
$$\text{with} \quad \#\{r : j_r = 1\} = \#\{r : j_r = 2\} \text{ and } j_1 = 1.$$

Because $z_{3-j_1} \cdot .. \cdot z_{3-j_n}$ is the complex conjugate of $z_{j_1} \cdot .. \cdot z_{j_n}$, this means that the space of SO invariants on level n in $T(\mathbb{C}^2)$ is spanned freely by the set of

$$\operatorname{Re}(z_{j_1} \cdot .. \cdot z_{j_n}) \quad \text{and} \quad \operatorname{Im}(z_{j_1} \cdot .. \cdot z_{j_n})$$
$$\text{with} \quad \#\{r : j_r = 1\} = \#\{r : j_r = 2\} \text{ and } j_1 = 1.$$

This is an expression for a basis of the SO invariants in terms of real combinations of basis elements of the tensor space. They thus form a basis for the SO invariants for the free *real* vector space on the same set, namely level n of $T(\mathbb{R}^2)$. □

I call the elements of this basis of real rotational invariants *raw rotational invariants*. Note that they each take values in a single level of tensor space (level n) and that that level is always even, due to (2.7).

2.3.1 Numerical calculation of raw and reduced invariants

I have implemented the calculation of the invariants of a path efficiently from its signature because of a deep learning research application where people wanted to do this repeatedly. This is part of the `iisignature` library. Here I make an observation which helped.

Consider the elements of level $2k$ of the signature as a zero-based array of length 2^{2k}. Then the binary expansion of the index of an element indicates the word which the signature element represents. For example the signature element for the word **2122** will be element $1011_2 = 11_{10}$ of level 4.

The terms in the raw invariants which come from the imaginary part will be each of the length-$2k$ strings which have an odd number of **1**s and an odd number of **2**s in, each with a factor of 1 or -1. Such invariants will consist of signature elements whose indices have an odd number of ones in their binary expansion, i.e. are *odious numbers* in the terminology of [BCG82]. I call them *odious invariants*. The value of an odious invariant on a path is negated when that path is reflected.

The terms in the raw invariants which come from the real part will be each of the length-$2k$ strings which have an even number of x_1s and an even number of x_2s in, each with a factor of 1 or -1. Such invariants will consist of signature elements whose indices have an even number of ones in their binary expansion, i.e. are *evil numbers*. I call them *evil invariants*. The evil invariants are invariant under reflections as well as rotation.

There are $\binom{2k}{k}$ raw invariants at level $2k$, of which half are odious and half are evil. For example, at level two there are two raw invariants: $\mathbf{12} - \mathbf{21}$ which is double the signed area and is odious, and $\mathbf{11} + \mathbf{22}$ which is the squared length of the total displacement and is evil. At level four there are six raw invariants, three odious

$$\begin{aligned}
-\mathbf{1112} - \mathbf{1121} + \mathbf{1211} - \mathbf{1222} + \mathbf{2111} - \mathbf{2122} + \mathbf{2212} + \mathbf{2221}\\
-\mathbf{1112} + \mathbf{1121} - \mathbf{1211} - \mathbf{1222} + \mathbf{2111} + \mathbf{2122} - \mathbf{2212} + \mathbf{2221}\\
\mathbf{1112} - \mathbf{1121} - \mathbf{1211} - \mathbf{1222} + \mathbf{2111} + \mathbf{2122} + \mathbf{2212} - \mathbf{2221}
\end{aligned}$$

and three evil

$$\begin{aligned}
\mathbf{1111} - \mathbf{1122} + \mathbf{1212} + \mathbf{1221} + \mathbf{2112} + \mathbf{2121} - \mathbf{2211} + \mathbf{2222}\\
\mathbf{1111} + \mathbf{1122} - \mathbf{1212} + \mathbf{1221} + \mathbf{2112} - \mathbf{2121} + \mathbf{2211} + \mathbf{2222}\\
\mathbf{1111} + \mathbf{1122} + \mathbf{1212} - \mathbf{1221} - \mathbf{2112} + \mathbf{2121} + \mathbf{2211} + \mathbf{2222}.
\end{aligned}$$

As an aside, in deep learning applications, we care about the scaling of our representations as explained in subsection 5.4.6. Each raw invariant is a sum of 2^{m-1} elements in level m of the signature, with some negated. This may mean that the appropriate scaling of these data as input to a machine learning algorithm may differ by this factor from the appropriate scaling of the signature.

As described in [Die13], due to the shuffle-product property of signatures, the values of some rotational invariants are known given the values of others at lower levels. Just as the log signature is useful as a minimal set of features from the signature, it may be useful to have a minimal representation of the rotational invariants, that is, a minimal representation of the signature information assigned to an equivalence class under rotation. We do not have a method to do this directly. We can achieve it at each level by finding the known invariants as shuffle products of lower raw invariants, and quotienting the span of the raw invariants by their span.

If $\mathcal{A}$ is the set of raw invariant vectors at level $2k$, and $\mathcal{B}$ is the set of known invariants at level $2k$, we can find a basis for the quotient as follows. First find a basis $\mathcal{B}'$ for the known invariants using the singular value decomposition or QR factorisation with pivoting. Then we project each element of a of $\mathcal{A}$ away from each element b of $\mathcal{B}'$ by replacing a by $a - b \cdot a$. Finally, we get a basis for the projected as using singular value decomposition or QR factorisation with pivoting again.

The shuffle product of two raw invariants is odious if exactly one of them is odious, and evil otherwise. Thus, by keeping track of odious and evil invariants everywhere, we can apply the procedure in the previous paragraph separately for the two cases. This means that although at each level the procedure must happen twice, the vectors concerned can be compressed to half as long, $\mathcal{A}$ is half the size, and $\mathcal{B}$ is about half the size in each case, which is a major time saving.

Unlike the raw invariants, the reduced invariants as returned by `iisignature` are unit vectors in (the dual of) each level of the signature, so it is plausible that the same scaling used for a calculation with the signature could be used with the reduced invariants.

Chapter 3

Areas of Areas

To signature elements ϕ and ψ, we can associate paths $t \mapsto \langle \phi, X^{\gamma}_{0,t} \rangle$ and $t \mapsto \langle \psi, X^{\gamma}_{0,t} \rangle$ given a path γ in $\mathbb{R}^d$. Then we can consider the signed area of the 2D path whose parametric coordinates are the values of these signature elements.

$$\begin{aligned} \mathsf{Area}(f, g) &:= \int df dg - \int dg df \\ f_t &= \langle \phi, X^{\gamma}_{0,t} \rangle \\ g_t &= \langle \psi, X^{\gamma}_{0,t} \rangle. \end{aligned}$$

We will shortly define an operation area with the following property, and consider its properties.

$$\mathsf{Area}(f, g)_t = \langle \mathsf{area}(\phi, \psi), X^{\gamma}_{0,t} \rangle.$$

In particular, if we start with the single letters and apply area between the terms we have, what elements of the signature can we span – alternatively, how much of the tensor algebra $T(\mathbb{R}^d) = \mathbb{R}\langle \Sigma \rangle$ can we span?

Definition 19. *Given two words/monomials ϕ and ψ, where ψ is not the empty word, and where the letters in ψ are given by $\psi = \psi_1 \dots \psi_m$, their* right half shuffle *is*

$$\phi \succ \psi := (\phi \sqcup\!\sqcup \psi_1..\psi_{m-1})\psi_m.$$

This is extended bilinearly to polynomials ϕ and ψ, as long as $\langle \epsilon, \psi \rangle = 0$. [1]

For example, $\mathbf{12} \succ \mathbf{3} = \mathbf{123}$, $\mathbf{3} \succ \mathbf{12} = \mathbf{312} + \mathbf{132}$ and $\mathbf{1} \succ \mathbf{1} = \mathbf{11}$. Note that the symmetrization of $\succ$ gives the shuffle product

$$a \succ b + b \succ a = a \sqcup\!\sqcup b,$$

which is why $\succ$ is called *half-shuffle*.

Now for monomials ϕ, ψ, ξ of order n_ϕ, n_ψ, n_ξ,

$$(\phi \succ \psi) \succ \xi + (\psi \succ \phi) \succ \xi = \phi \succ (\psi \succ \xi).$$

So nonempty words with $\succ$ form a (left) Zinbiel algebra.In fact, they generate the free Zinbiel algebra on $\mathbb{R}^d$, see Theorem 1.8 of [Lod95] or section 7 of [Lod01]. This

[1]The special case of the empty word ϵ is a bit of a pain but not enlightening. We do not need it here.

is in fact shown around page 19 of [Sch58], where $a \top b$ is effectively our $b \succ a$, $\overline{\top}$ becomes the shuffle product, and all words are reversed.

It is clear that starting from single letters and using $\succ$ every nonempty word can be made, because concatenating a letter l onto the end of a word w is the same as $w \succ l$.

We are interested in the anti-symmetrization of $\succ$.

Definition 20. *Define*

$$\mathsf{area}(a, b) := a \succ b - b \succ a.$$

Remark 21. *The operation* area *is anti-commutative but it is* not *a Lie product, since it does not satisfy the Jacobi identity. Indeed,*

$$\begin{aligned}\mathsf{area}(\mathbf{1}, \mathsf{area}(\mathbf{2}, \mathbf{3})) &= \mathsf{area}(\mathbf{1}, \mathbf{23} - \mathbf{32}) \\ &= \mathbf{123} + \mathbf{213} - \mathbf{132} - \mathbf{312} - \mathbf{231} + \mathbf{321},\end{aligned}$$

but

$$\mathsf{area}(\mathsf{area}(\mathbf{1}, \mathbf{2}), \mathbf{3}) + \mathsf{area}(\mathbf{2}, \mathsf{area}(\mathbf{1}, \mathbf{3})) = 2\mathbf{123} - 2\mathbf{132}.$$

Definition 22. *Define*

$$\begin{aligned} P^{(1)} &:= \Sigma = \{\mathbf{1}, \dots, \mathbf{d}\} \\ P^{(n+1)} &:= \{\mathsf{area}(\phi, \psi) : \phi, \psi \in \bigcup_{i=1}^{n} P^{(i)}\} \\ P &:= \bigcup_{n=2}^{\infty} P^{(n)}. \end{aligned}$$

This P is everything you can get with the area operation, the areas-of-areas, and it is the subject of our interest.

3.1 Linear span of P: upper bound

Definition 23. *Let A_d be those elements of $T(\mathbb{R}^d)$ which can be written as a concatenation of some element $x \in T(\mathbb{R}^d)$ and $(ij - ji)$ for $i, j \in \{\mathbf{1}, \dots, \mathbf{d}\}$.*

For example, A_2 contains $(3\mathbf{1} - \mathbf{12})(\mathbf{12} - \mathbf{21}) = 3\mathbf{112} - 3\mathbf{121} + \mathbf{1221} - \mathbf{1212}$ but it does not contain $\mathbf{1112}$. The order of the choice of i and j only affects sign, and to get a nonzero element i must not equal j. If $m \geq 2$ then level m is a d^m-dimensional space and the elements of A_d in level m form a $d^{m-2}\binom{d}{2}$-dimensional subspace.

Conjecture 24. $\operatorname{span} P = A_d$.

Intuitively, $\operatorname{span} P$ is subspace of tensor space constructed from letter building blocks and a slightly weird nonassociative operation, whilst A_d is a simple algebraic description of a subspace. This conjecture would explain the nature of the former in a nice way. It is a bit like the way a Hall basis explains the nature of the free Lie algebra, which is everything you can get from letter building blocks and the Lie bracket operation, but the content of the answer is simpler. The result of this section is one direction of Conjecture 24. In the next section we prove the conjecture for the case $d = 2$.[2]

Lemma 25. *Let i, j, k and l be letters. If $X = x(ij - ji)$ and $Y = y(kl - lk)$, then* $\mathsf{area}(X, Y) \in A_d$.

Proof. We have

$$\begin{aligned}
\mathsf{area}(X,Y) &= X \succ y(kl - lk) - Y \succ x(ij - ji) \\
&= (X \sqcup\!\sqcup yk)l - (X \sqcup\!\sqcup yl)k - (Y \sqcup\!\sqcup xi)j + (Y \sqcup\!\sqcup xj)i \\
&= (X \sqcup\!\sqcup y)kl + (xi \sqcup\!\sqcup yk)jl - (xj \sqcup\!\sqcup yk)il \\
&\quad - (X \sqcup\!\sqcup y)lk - (xi \sqcup\!\sqcup yl)jk + (xj \sqcup\!\sqcup yl)ik \\
&\quad - (Y \sqcup\!\sqcup x)ij - (yk \sqcup\!\sqcup xi)lj + (yl \sqcup\!\sqcup xi)kj \\
&\quad + (Y \sqcup\!\sqcup x)ji + (yk \sqcup\!\sqcup xj)li - (yl \sqcup\!\sqcup xj)ki \\
&= (X \sqcup\!\sqcup y)(kl - lk) - (Y \sqcup\!\sqcup x)(ij - ji) \\
&\quad + (xi \sqcup\!\sqcup yk)(jl - lj) - (xi \sqcup\!\sqcup yl)(jk - kj) \\
&\quad + (xj \sqcup\!\sqcup yl)(ik - ki) - (xj \sqcup\!\sqcup yk)(il - li) \\
&\in A_d.
\end{aligned}$$

□

Lemma 26. *If $X, Y \in A_d$, then* $\mathsf{area}(X, Y) \in A_d$.

Proof. X and Y are linear combinations of expressions to which the previous lemma can be applied. □

The letters are not in A_d, but bracketing with them is also fine:

[2] The preprint [DIM18], which was submitted to ArXiv just before I submitted this thesis, appears to prove this conjecture in general, using a different argument. I was not aware of it when working on this.

Lemma 27. *If $X \in A_d$ and $k \in \{\mathbf{1}, \dots, \mathbf{d}\}$, then* $\mathsf{area}(k, X)$ *and* $\mathsf{area}(X, k)$ *are in* A_d.

Proof. X is the sum of terms like $x(ij - ji)$ where i and j are letters. For such a term

$$\begin{aligned}
\mathsf{area}(x(ij-ji),k) &= (xij - xji) \succ k - k \succ (xij - xji) \\
&= xijk - xjik - (k \sqcup\!\sqcup xi)j + (k \sqcup\!\sqcup xj)i \\
&= xijk - xjik - xikj - (k \sqcup\!\sqcup x)ij + xjki + (k \sqcup\!\sqcup x)ji \\
&= xi(jk - kj) + xj(ki - ik) + (k \sqcup\!\sqcup x)(ji - ij) \qquad (3.1)\\
&\in A_d
\end{aligned}$$

$\mathsf{area}(X, k)$ being in A_d follows because area is linear. $\mathsf{area}(k, X)$ is minus $\mathsf{area}(X, k)$ and so is also in A_d which is a subspace. $\square$

Theorem 28. $\operatorname{span} P \subset A_d$

Proof. $P^{(2)}$ is exactly areas between pairs of letters, which are of the form $ij - ji$. These are in level 2 of A_d. Other elements of P are formed either as the area between two lower level elements of P or the area between a letter and a lower level element of P. These are all in A_d by induction, using Lemma 26 and Lemma 27. Thus $P \subset A_d$, and because A_d is a vector space, $\operatorname{span} P \subset A_d$. $\square$

We conjecture further that the span of left-bracketed areas of areas is the same as all areas of areas, which we show to be the case for $d = 2$ in the next section.

3.2 Linear span of P: two-dimensional case

We restrict attention in this section to the $d = 2$ case, and show that the linear span of P is in fact the whole of A_2. In fact, the linear span of just the area expressions which are wholly nested is the whole of A_2. Aside from swapping the order in the innermost bracket, which must contain a $\mathbf{1}$ and a $\mathbf{2}$, such nested area expressions are linearly independent.

I use the permutation convention under which "do (13) and then do (12)" is $(12)(13) = (132)$, and I denote the identity permutation by id. If b is an element of the group algebra $\mathbb{R}S_n$ and $\sigma \in S_n$ then I denote the coefficient of σ in b as $b(\sigma)$.

Lemma 29. *Let n be a positive integer, and consider the following element of $\mathbb{R}S_n$, the group algebra of the symmetric group.*

$$\phi_n = 2\mathsf{id} + (21) + (321) + \cdots + (n \ldots 1) \tag{3.2}$$

Then ϕ_n is a unit, i.e. there exists an element ϕ_n^{-1} such that $\phi_n \phi_n^{-1} = \phi_n^{-1} \phi_n = \mathsf{id}$.

In particular, if f is a function from permutations to a real vector space,

$$\sum_{\sigma' \in S_n} \phi_n^{-1}(\sigma') \sum_{\sigma \in S_n} \phi_n(\sigma) f(\sigma' \sigma) = f(\mathsf{id}). \tag{3.3}$$

The proof of this was explained to me in detail by Darij Grinberg in [Gri18]. In fact, all the values of ϕ_n^{-1} are rational and so the Lemma can be stated and proved just the same for $\mathbb{Q}S_n$ instead of $\mathbb{R}S_n$, but we do not need this strengthening.[3]

Proof. The element $\psi_n := (\phi_n - \mathsf{id})$ of $\mathbb{R}S_n$ is considered in many places, being known by such names as the *top-to-random shuffle*, or the (transition matrix of the) *Tsetlin library*. The eigenvalues of ψ_n (i.e. of the linear map from $\mathbb{R}S_n$ to itself defined by multiplication by ψ_n) are known to be $0, 1, \ldots, n-2, n$; for example they are specified in Theorem 2.2 of [Gar12]. This goes back to [DFP92]. Since these eigenvalues of ψ_n do not include -1, the eigenvalues of multiplication by $\phi_n = \mathsf{id} + \psi_n$ do not include 0, and so the multiplication is a linear endomorphism of $\mathbb{R}S_n$. This means that ϕ_n must be a unit. $\square$

For example, $\phi_2^{-1} = \frac{1}{3}[2e - (12)]$ and $\phi_3^{-1} = \frac{1}{8}[5e + (23) - 3(12) - 3(132) + (123) + (13)]$.

Definition 30. *If $w = l_1 \ldots l_k$ is a word, we define $\underline{\mathsf{area}}(w)$ to be the left-bracketing expression*

$$\mathsf{area}(\ldots \mathsf{area}(\mathsf{area}(\mathsf{area}(l_1, l_2), l_3), l_4), \ldots, l_k). \tag{3.4}$$

Theorem 31. *For k a nonnegative integer, the linear span of the elements $\underline{\mathsf{area}}(\mathbf{12}w)$ where w ranges over words in $\{\mathbf{1}, \mathbf{2}\}$ of length k is the whole of level $k+2$ of A_2, that is the span of elements $v(\mathbf{12} - \mathbf{21})$ where v also ranges over words in $\{\mathbf{1}, \mathbf{2}\}$ of length k.*

Proof. For $k = 0$, this is clear because $\underline{\mathsf{area}}(\mathbf{12}) = \mathbf{12} - \mathbf{21}$. Assume the statement is true for all $k \leq n$.

[3] I used the GAP computer algebra system to test some of this out, where working with $\mathbb{Q}S_n$ is the sensible way to proceed. That is why I asked the question in that setting.

Let $wj_1 = j_{n+1}j_n..j_2j_1$ be a word of length $n+1$. We see from (3.1) that

$$\begin{aligned}\mathsf{area}(w(\mathbf{12}-\mathbf{21}), j_1) &= -(wj_1 + j_1 \sqcup\!\sqcup\, w)(\mathbf{12}-\mathbf{21})\\ &= -(2wj_1 + j_1 \succ w)(\mathbf{12}-\mathbf{21}) \qquad (3.5)\end{aligned}$$

In other words,

$$\begin{aligned}\mathsf{area}(j_{n+1}..j_2(\mathbf{12}-\mathbf{21}), j_1) &= -\big(2j_{n+1}..j_1 + j_1 \succ j_{n+1}..j_2\big)(\mathbf{12}-\mathbf{21})\\ &= -\big(\sum_{\sigma\in S_{n+1}} \phi_{n+1}(\sigma) j_{\sigma(n+1)}\dots j_{\sigma(1)}\big)(\mathbf{12}-\mathbf{21}) \qquad (3.6)\end{aligned}$$

Summing both sides with each j_l replaced by $j_{\sigma'(l)}$ where σ' varies over the weighted permutations in ϕ^{-1}_{n+1}, and negating both sides, gives

$$\begin{aligned}&-\sum_{\sigma'\in S_{n+1}} \phi^{-1}_{n+1}(\sigma')\mathsf{area}(j_{\sigma'(n+1)}\dots j_{\sigma'(2)}(\mathbf{12}-\mathbf{21}), j_{\sigma'(1)})\\ &= \sum_{\sigma'\in S_{n+1}} \phi^{-1}_{n+1}(\sigma')\big(\sum_{\sigma\in S_{n+1}} \phi_{n+1}(\sigma) j_{\sigma'(\sigma(n+1))}\dots j_{\sigma'(\sigma(1))}\big)(\mathbf{12}-\mathbf{21})\\ &= \sum_{\sigma'\in S_{n+1}} \phi^{-1}_{n+1}(\sigma')\big(\sum_{\sigma\in S_{n+1}} \phi_{n+1}(\sigma) j_{(\sigma'\sigma)(n+1)}\dots j_{(\sigma'\sigma)(1)}\big)(\mathbf{12}-\mathbf{21})\end{aligned}$$

(using (3.3))

$$= j_{n+1}..j_1(\mathbf{12}-\mathbf{21}) = wj_1(\mathbf{12}-\mathbf{21}).$$

We have written the generic basis element $wj_1(\mathbf{12}-\mathbf{21})$ of level $n+3$ of A_2 as a linear combination of elements $\mathsf{area}(v, j_1)$ for v in level $n+2$ of A_2 and $j_1 \in \{\mathbf{1}, \mathbf{2}\}$. By hypothesis, any such v is in the span of left-bracketed areas. Thus our generic basis element of level $n+3$ of A_2 is in the span of left-bracketed areas and the statement is true for $k = n+1$. □

In summary, for the $d = 2$ case, within each level, we have the combined inclusions

$$\operatorname{span} P \overbrace{\subset}^{\text{Theorem 28}} A_2 \underbrace{=}_{\text{Theorem 31}} \operatorname{span}\{\underline{\mathsf{area}}(\mathbf{12}w) \mid w \text{ word}\} \overbrace{\subset}^{\text{trivial}} \operatorname{span} P$$

so A_2 *is* span P, the span of all area expressions, and we have that these left-bracketed area expressions form a basis.

Chapter 4

Calculation of signatures

4.1 Introduction

In this chapter we present algorithms for efficiently calculating the signatures and log signatures of piecewise linear paths. We also talk about efficiently backpropagating derivatives through these functions. This is useful for any machine learning task where these calculations need to be performed repeatedly. This functionality is implemented in the `iisignature` Python package.

The focus of the `iisignature` package is the calculation of the signature for piecewise-linear paths in fixed-dimensional spaces. In these relatively low dimensional spaces, paths typically move in all their dimensions, so only rarely will elements of the signature be zero. We call this the *dense* case. We study the mathematical properties of the free Lie algebra to implement a range of algorithms. We also benchmark the performance of these algorithms, and provide an efficient open-source implementation.

An existing open-source library for calculating signatures is the `esig` package from CoRoPa[Lyo+10]. However, this package is optimized to operate efficiently on another type of path: ones that live in high dimensional spaces, but that only move in certain combinations of input dimensions. Most of the elements of the signatures are zero. We call this the *sparse* case.

4.2 Signatures

Calculating the signature of a path can be done inductively relying on the following two rules.

- If γ is a straight line defined on the interval $[a, b]$ then its signature as a function on words is
$$X^{\gamma}_{a,b}(i_1 i_2 \dots i_m) = \frac{1}{m!}\prod_{j=1}^{m}(\gamma_{i_j}(b) - \gamma_{i_j}(a)). \tag{4.1}$$
Grouped by levels, using $x = \gamma(b) - \gamma(a)$ as the displacement, the signature looks like
$$\left(1, x, \frac{x \otimes x}{2!}, \frac{x \otimes x \otimes x}{3!}, \dots\right) \tag{4.2}$$
where $\otimes$ is the tensor product. Alternatively, if each level is thought of as a vector of numbers, this formula should be read with $\otimes$ denoting the Kronecker product.

- If $a < b < c$ then the result (from [Che58]) known as **Chen's identity** states that

$$X^{\gamma}_{a,c}(i_1 i_2 \dots i_m) = \sum_{j=0}^{m} X^{\gamma}_{a,b}(i_1 i_2 \dots i_{j-1}) X^{\gamma}_{b,c}(i_j i_{j+1} \dots i_m). \tag{4.3}$$

Grouped by levels, this signature looks like

$$\begin{aligned}\Big(1, X^{(1)}_{a,c}, X^{(2)}_{a,c}, \dots\Big) = \Big(1, X^{(1)}_{a,b} + X^{(1)}_{b,c}, X^{(2)}_{a,b} + X^{(1)}_{a,b} \otimes X^{(1)}_{b,c} + X^{(2)}_{b,c}, \\ X^{(3)}_{a,b} + X^{(2)}_{a,b} \otimes X^{(1)}_{b,c} + X^{(1)}_{a,b} \otimes X^{(2)}_{b,c} + X^{(3)}_{b,c}, \dots\Big)\end{aligned} \tag{4.4}$$

When calculating the signature of a path given as a series of straight-line displacements, we start with the signature of the first displacement (calculated from (4.1)) and step-by-step concatenate on the signature of each succeeding displacement using (4.3).

Level m of the signature contains d^m values. Calculating it for a displacement using (4.1) takes $d + d^m$ multiplications beyond what has already been calculated for lower levels. However, in the signature of a straight line, each level is a symmetric tensor and so level m only contains $\binom{d+m-1}{m}$ distinct values, using the formula for unordered sampling with replacement. An alternative, more complicated, method that takes account of this redundancy exists. Only $d + \binom{d+m-1}{m}$ multiplications are required. Implementing it showed it to be slower, so `iisignature` does not use this idea.

4.3 Log Signatures directly

The log signature of a straight line displacement is just the displacement itself in level 1, and zero in every other level. The log signature of the concatenation of two paths is the Baker-Campbell-Hausdorff (BCH) product of the log signatures of the two paths. The direct method for calculating the log signature relies on being able to transform the log signature of a path given in terms of one of the bases above to the log signature of that path concatenated with a fixed line segment, achieved using the BCH product.

The BCH product is an infinite series in bracketed expressions in two indeterminates, which has can be formulated in different equivalent ways. The most straightforward ways express all brackets in the form of some Hall basis of the free

Lie algebra of $\mathbb{R}^2$. For example, using the Lyndon basis:

$$\text{bch}(a,b) = a + b + \tfrac{1}{2}[a,b] + \tfrac{1}{12}[a,[a,b]] + \tfrac{1}{12}[[a,b],b] + \tfrac{1}{24}[a,[[a,b],b]] + \dots.$$

The coefficients in this expansion up to terms of depth twenty have been calculated and distributed by Fernando Casas and Ander Murua at [CM], using their method described in [CM09]. We distribute their file as part of `iisignature`, and read it when necessary.

We can compute the Lie bracket of each pair of basis elements as a combination of other basis elements, and therefore, given two log signatures as combinations of basis elements (the second known to be just a displacement) we can find the expanded expression of their BCH product as a combination of basis elements. By doing this with indeterminates, the library develops an internal representation.

As an example, in the case where the Lyndon basis is used, and we are concerned with two dimensions up to level two, a log signature looks like

$$a_0\mathbf{1} + a_1\mathbf{2} + a_2\mathbf{12}$$

The inductive step of the algorithm to accumulate log signatures by adding linear segments for $d = m = 2$ is shown in Figure 4.1.

```
def F22(a, b):
    # Construct monomials of log signature a
    #  and displacement b
    t[0] = b[1] * a[0]
    t[1] = b[0] * a[1]
    # Extend log signature in-place
    a[2] += t[0] / 2
    a[2] -= t[1] / 2
    a[0:2] += b[:]
```

Figure 4.1: Algorithm to accumulate a new displacement into a log signature in the Lyndon basis with $d = 2$ and $m = 2$.

If we go up to level 3, a log signature looks like

$$a_0\mathbf{1} + a_1\mathbf{2} + a_2\mathbf{12} + a_3\mathbf{112} + a_4\mathbf{122},$$

with the final algorithm being as shown in Figure 4.2.

```
def F23(a, b): # Log signature a and displacement b
    # Calculate monomials of a and b
    t[0]   += b[1] * a[0] # Order 2 monomials
    t[1]   += b[1] * a[2]
    t[2]   += b[0] * a[1]
    t[3]   += b[0] * a[2]
    t[4]   += b[1] * t[0] # Order 3 monomials
    t[5]   += b[0] * t[0] #  calculated from
    t[6]   += b[1] * t[2] #  t[i], i<=4
    t[7]   += a[0] * t[0]
    t[8]   += a[1] * t[0]
    t[9]   += b[0] * t[2]
    t[10]  += a[0] * t[2]
    t[11]  += a[1] * t[2]
    # Extend log signature in-place
    a[2]   +=  t[0]/2 - t[2]/2
    a[3]   += -t[3]/2 - t[5]/12 + t[7]/12 +
                        t[9]/12 - t[10]/12
    a[4]   +=  t[1]/2 + t[4]/12 - t[6]/12 -
                        t[8]/12 + t[11]/12
    a[0:2] +=  b[:]
```

Figure 4.2: Algorithm to accumulate a new displacement into a log signature in the Lyndon basis with $d = 2$ and $m = 3$.

These functions have a lot of common structure. First a sequence of monomials in the input elements are constructed in the temporary array t. Higher order monomials are calculated inductively from other elements of t to deduplicate the necessary multiplications. Then some members of a are incremented by some multiples of some of the temporary variables. Then the first d elements of a are incremented by all elements of b. Exactly which is given by the `FunctionData` structure. In general these functions are long and branching-free. The variable a is modified in-place to produce the log signature of the extended path.

The basis (of the free Lie algebra on 2 symbols) used to express the BCH formula does not change the code we get, because the various equivalent bracketed expressions come to the same thing when they have been multiplied out. We use the Lyndon basis because it has slightly fewer terms, as [CM09] describes and partially explains. This choice is independent of the choice of basis (of the free Lie algebra on d symbols) in which the log signature is expressed. In general, we end up with fewer terms and a slightly faster calculation when the Lyndon basis is used for the log signature.

4.4 Log Signatures from Signatures

A simple method, which we call the `"S"` method for calculating the log signature of a path is to calculate its signature first, and then convert to the log signature. The first step in doing the conversion is taking the logarithm itself in tensor space. This explicitly uses the formula (1.5) where n only needs to go as high as the required level, and the power is in the concatenation product. This results in the log signature as an element of tensor space (which means it is as long as a signature), which is returned when `logsig` is called with the `"X"` (*expanded*) method. The exact order of evaluation of formula (1.5) for best efficiency which we use is one which was suggested by Mike Giles[Gil17].

To express this Lie element into a specified basis, we need to project it. We calculate a projection explicitly. There are known explicit forms for projections, for example the map given by the Dynkin-Specht-Wever lemma directly ([Wil12]), which requires more operations. The `prepare` function calculates a projection upfront.

Given the bracketed expression of a basis element with m letters, we can easily find its expression in expanded space, by multiplying out the brackets. For example, $[[\mathbf{1},\mathbf{3}],\mathbf{3}]$ is $\mathbf{133} - 2\,\mathbf{313} + \mathbf{331}$. This gives us the full matrix M_m to transform each level of the log signature to its expanded version. Each column of M_m is labelled with a basis element, and each row is labelled with one of the d^m words of length d. To compress level m a given expanded log signature x_m to its value c_m in terms of a basis, we just need to solve a least squares problem $M_m c_m = x_m$. This problem is a very overdetermined system which is known to have an exact answer, up to rounding considerations. M_m is tall and skinny.

The words occurring in the terms of the expansion of such a bracketed expression are anagrams of the foliage of the expression. In the terminology of [Reu94], these operations preserve the *fine homogeneity*. In that same example, for instance, **133**, **313** and **331** are anagrams of **133**. This leads to a lot of sparsity in the matrix M_m. Permuting the rows and columns to gather anagrams makes M_m be a block

letter frequencies	number of classes	signature elements in each	log signature elements in each	total log signature elements
$\{4,3,3\}$	3	4200	420	1260
$\{4,4,2\}$	3	3150	312	936
$\{5,3,2\}$	6	2520	252	1512
$\{5,4,1\}$	6	1260	126	756
$\{6,2,2\}$	3	1260	124	372

Table 4.1: The sizes of the largest anagram classes for level 10 of $d = 3$ in decreasing order of number of log signature elements. Many more such statistics have been tabulated in [Blü04].

diagonal matrix. We can save time doing the transformation by solving a separate linear system for each equivalence class of anagrams of words of length m.

For the standard Hall basis, this is exactly the procedure which we follow. In `prepare`, we determine all the mapping matrices between anagram classes of the log signature and its expansion, and then we calculate all their Moore-Penrose pseudoinverses, so that solving the systems is just a matrix multiplication. The number of words in an anagram set containing m letters where the frequency of the ith letter is n_i is given by a multinomial coefficient $\frac{m!}{n_1!\dots n_d!}$. The number of Lie basis elements in an anagram set is given by the second Witt formula of Satz 3 of [Wit37] as

$$\ell_m(n_1,\dots,n_d) = \frac{1}{m}\sum_{\delta|n_i} \frac{\mu(\delta)(\frac{m}{\delta})!}{(\frac{n_1}{\delta})!\dots(\frac{n_d}{\delta})!}, \tag{4.5}$$

where δ ranges over all common factors of the n_i and μ is the Möbius function. In the simple special case that the words have m distinct letters, there are $m!$ words and $(m-1)!$ basis elements. In the Lyndon case, this formula makes sense because the Lyndon words in such a set of $m!$ words are just all that begin with the lowest letter. Typically the largest anagram sets are the ones with about the same number of each letter. For them, (4.5) is just $\frac{1}{m}$ times the number of words in the set because 1 is the only value of δ. For example, looking at level 10 for a 3-dimensional path, the signature has 59049 elements and the log signature 5880, and there are 63 anagram classes.[1] The 12 most balanced anagram classes account for 3708 elements of the log signature, or 63.1% of it.

The big anagram classes account for most of the runtime when projecting to

[1]The count is $63 = \binom{10+3-1}{10} - 3$ using the formula for unordered sampling with replacement and the fact that no basis element above level 1 has only one distinct letter in it.

the log signature: multiplying a 420×4200 matrix by a 4200-vector takes 80% more multiplications than multiplying a 312×3150 matrix by a 3150-vector and so on.

4.4.1 Lyndon case

If the Lyndon basis is required, then we have a more efficient implementation, which depends on a special property it has. Recall the notation P_a (section 1.4 above and pages 89–91 of [Reu94]) for the Lie polynomial corresponding to the Hall word a, i.e. the polynomial you get by multiplying out the bracketed expression corresponding to the unique basis element whose foliage is a. Recall also that in the Lyndon basis the Lyndon words *are* the Hall words. We have

Theorem 32 (Theorem 5.1 of [Reu94]). *The set of Lyndon words, ordered alphabetically, is a Hall set. The corresponding Hall basis has the following triangularity property: for each word $w = l_1 \dots l_n$ written as a decreasing product of Lyndon words, the polynomial $P_w = P_{l_1} \dots P_{l_n}$ is equal to w plus a $\mathbb{Z}$-linear combination of greater words.*

The simplest case of the final statement, where w is itself a single Lyndon word, gives the following useful fact. When the bracketed expression corresponding to a Lyndon word is expanded and terms are collected and ordered in alphabetical order of the word, the first term will be the Lyndon word itself, with coefficient 1. (For an example, consider the Lyndon word **133**; its bracketed expression is $[[\mathbf{1},\mathbf{3}],\mathbf{3}]$ and we saw earlier that this expands to $\mathbf{133} - 2\,\mathbf{313} + \mathbf{331}$.) This means that the tall skinny matrix M_m is lower triangular, as are its anagram blocks. If we take such a block and remove all the rows corresponding to words which are not Lyndon, we are left with the mapping from an anagram class in the compressed log signature to same Lyndon word elements of the expanded signature. It is a square lower triangular matrix with ones on the diagonal. We can now solve the system directly in many fewer operations, with just addition and multiplication, just looking at the Lyndon word elements of the expanded signature. `prepare` determines the necessary indices and matrices, and `logsig` does the solving.

For example, in level 4 on 3 dimensions, the following are the three basis elements which contain two **1**s, a **2** and a **3**:

$$[\mathbf{1},[\mathbf{1},[\mathbf{2},\mathbf{3}]]] = \mathbf{1123} - \mathbf{1132} - 2\,\mathbf{1231} + 2\,\mathbf{1321} + \mathbf{2311} - \mathbf{3211}$$
$$[\mathbf{1},[[\mathbf{1},\mathbf{3}],\mathbf{2}]] = \mathbf{1132} - \mathbf{1213} + \mathbf{1231} - \mathbf{1312} - \mathbf{1321} + \mathbf{2131} - \mathbf{2311} + \mathbf{3121}$$
$$[[\mathbf{1},\mathbf{2}],[\mathbf{1},\mathbf{3}]] = \mathbf{1213} - \mathbf{1231} - \mathbf{1312} + \mathbf{1321} - \mathbf{2113} + \mathbf{2131} + \mathbf{3112} - \mathbf{3121}$$

The matrix corresponding to these looks as follows

$$\begin{array}{cccc}
[1,[1,[2,3]]] & [1,[[1,3],2]] & [[1,2],[1,3]] & \\
\left(\begin{array}{c} 1 \\ -1 \\ 0 \\ -2 \\ 0 \\ 2 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ -1 \end{array}\right. &
\begin{array}{c} 0 \\ 1 \\ -1 \\ 1 \\ -1 \\ -1 \\ 0 \\ 1 \\ -1 \\ 0 \\ 1 \\ 0 \end{array} &
\left.\begin{array}{c} 0 \\ 0 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 0 \\ 0 \\ 1 \\ -1 \\ 1 \end{array}\right) &
\begin{array}{c} \mathbf{1123} \\ \mathbf{1132} \\ \mathbf{1213} \\ \mathbf{1231} \\ \mathbf{1312} \\ \mathbf{1321} \\ \mathbf{2113} \\ \mathbf{2131} \\ \mathbf{2311} \\ \mathbf{3112} \\ \mathbf{3121} \\ \mathbf{3211} \end{array}
\end{array}$$

and when we restrict to Lyndon words (which in general are not the first rows) we get a matrix which has $m = 4$ times fewer rows, and is a lower triangular square matrix with ones on the diagonal.

$$\begin{array}{cccc}
[1,[1,[2,3]]] & [1,[[1,3],2]] & [[1,2],[1,3]] & \\
\left(\begin{array}{c} 1 \\ -1 \\ 0 \end{array}\right. &
\begin{array}{c} 0 \\ 1 \\ -1 \end{array} &
\left.\begin{array}{c} 0 \\ 0 \\ 1 \end{array}\right) &
\begin{array}{c} \mathbf{1123} \\ \mathbf{1132} \\ \mathbf{1213} \end{array}
\end{array}.$$

If this matrix is called M' we can solve the equation $M'c' = x'$ directly using

$$c'_1 = x'_1 \qquad c'_2 = x'_2 - (-1\,c'_1) \qquad c'_3 = x'_3 - (0\,c'_1 - 1\,c'_2).$$

4.5 Implementation

`iisignature` is a Python package which is built on `numpy`[WCV11], which is ubiquitous for dealing with numerical data in Python. The Python ecosystem is very commonly used for deep learning. It is implemented as a C++ extension.

There is a single `.cpp` file which defines the whole interface with Python. The mathematical functionality resides in header files. This *unity build* structure reduces the time to build the whole library, which matters to users, at the cost of incremental build time.

4.5.1 Signatures

We store signatures during the calculation with each level in a contiguous block of memory, which means that accesses are efficient. We start with the signature of the first displacement and step-by-step concatenate on the signature of each succeeding displacement. The concatenation is done in place, but in simple cases this doesn't seem to make a difference in performance.

We also wrote an implementation of the signature calculation using a template metaprogramming style, where the dimension and level are template parameters, there is no heap memory allocation and all loops are constant length. We compared the methods and learnt that the performance is the same. Because we want to allow arbitrary calculations easily for the user, it is convenient not to code in this way inside iisignature.

4.5.2 Preparing the direct calculation of log signatures

The internal representation of the calculation required to convert the log signature of a path into the log signature of that path with a line segment concatenated on the end is stored in an instance of the `FunctionData` class. The calculation depends on the following concepts. The class `Input` represents an indeterminate, and a `Coefficient` is a polynomial in `Input`s Elements of the basis of the free Lie algebra we are using are represented by instances of the `BasisElt` class. These are created once for a whole calculation, and they all live together in memory controlled by a `BasisPool` which also remembers their order and those of their Lie products (Lie brackets) which happen to be `BasisElt`s. Elements of the free Lie algebra, Lie polynomials, are represented by the class `Polynomial`. In a similar way as [Lyo+10], we store the data of a `Polynomial` with each level separately, this speeds up the multiplication of two of them very much, because it becomes trivial to avoid trying to multiply terms whose combined level will exceed the level we are truncating at. The procedure calculates the BCH product of an arbitrary polynomial (one with a separate indeterminate for each basis element, representing an arbitrary log signature) and an arbitrary level-one polynomial (one which is just a separate indeterminate for each letter, representing the log signature of a single displacement) to produce a `Polynomial` which is exactly what the `FunctionData` needs to calculate.

type	algebraic structure	definition	instance represents
`double`	field (roughly)	C++ builtin	constant floating point real
`Input`	set	indeterminate	numeric input
`Coefficient`	semigroup ring	polynomial from `double`[{`Input`s}]	formula in terms of the inputs
`BasisElt`	set	(fixed but complicated)	element of the given basis of the FLA
`Polynomial`	free vector space augmented with Lie bracket	function from `BasisElt` to `Coefficient`	element of FLA

Table 4.2: Summary of the main object types for the Free Lie algebra (FLA) calculations

In the `"O"` (*object*) mode, the `prepare` function goes as far as computing this `FunctionData` object, and the `logsig` function follows its instructions for dealing with each displacement, using the function `slowExplicitFunction`.

4.5.3 On-the-fly machine code generation for the direct calculation

Code specifically compiled for the particular function is more efficient than following instructions given by the `FunctionData` object. Before this library, we wrote some code (described in [Rei15] and demonstrated at `https://github.com/bottler/LogSignatureDemo`) to generate C++ code which can be compiled to give efficient versions of this function. We learnt that while this method is very efficient, it is impractical for many realistic d and m because the function can easily get so large that compilers take unreasonably long times to compile it. Attempting to split them up only helps a small amount. Manual machine code generation avoids this delay. `iisignature` therefore provides the `"C"` method under which it compiles the `FunctionData` itself on-the-fly to machine code internally in a buffer in memory during `prepare`, and all `logsig` need do is run the compiled code for each displacement. This is implemented for x86 and x86-64 architectures, for Windows, Linux and Mac.

The logic for the compilation is in makeCompiledFunction.hpp. The Mem object represents a buffer for storing machine code, which is allocated in such a way that execution is enabled. The FunctionRunner object is constructed from a FunctionData and allocates a Mem and compiles the function into it, providing a go() function to run the compiled code. The actual compilation is done by the Maker class. The comments in that class explain what it is doing in terms of x86 and x86-64 machine code instructions. In the x86 case, we rely on SSE2 instructions for floating point arithmetic which enables the logic to be roughly the same as in the 64-bit case.

4.5.4 Projection from expanded log signature to a basis

The makeMappingMatrix function calculates the full matrix (in sparse form) to project from tensor space to the desired basis. The identification of anagram classes is performed in analyseMappingMatrixLevel. The makeSparseLogSigMatrices function identifies all the data needed to do the projection from this information. When, as often happens, a basis element's anagram class is a singleton, we can just read off its value from the expanded log signature without solving a system. In the standard Hall basis case, we need to calculate the Moore-Penrose pseudoinverses of the identified matrices, and we do this using numpy at the interface level. All the data to do this is stored in the object which prepare generates, and is available to be simply used by logsig.

4.6 Indicative timings

Using 64bit Python 3.5 on Ubuntu 16.04LTS with an AMD FX-8320, timings were taken for calculating 100 signatures of randomly generated paths with 100 timesteps in various different ways. We compare here a native python signature implementation using numpy, the calculation with iisignature version 0.20, and the calculation in the package esig.tosig of CoRoPa[Lyo+10], version 0.6.5. Both iisignature and esig use 64-bit floating point internally, but iisignature is taking and returning 32-bit floating point values in this example, whereas esig uses 64 bit throughout. It should be noted that esig was not specifically written to make this type of calculation fast, but for other types of flexibility (e.g. the sparse signature case). The dramatic difference shows the advantage of having code written specifically for the dense case.

(d,m)	(2,6)	(2,10)	(3,10)	(5,5)	(10,4)
Python native	10.56	78.66	2458.09	55.95	118.83
`iisignature.sig`	**0.02**	**0.27**	**10.78**	**0.29**	**0.61**
`esig.tosig.stream2sig`	1.98	55.51	3114.36	56.36	175.38

Table 4.3: Various signature calculation timings in seconds, for 100 random paths of 100 steps each in the given combinations of level and dimension

Calculating the log signature using the compiled method is quicker than calculating the signature for small depths, but for larger depths the signature becomes significantly faster. As the depth increases, the projection method therefore becomes the best method to obtain the log signature. For two dimensions, performance is shown in Table 4.4 and plotted in Figure 4.3, and for three dimensions performance is shown in Table 4.5 and plotted in Figure 4.4.

	level:	4	5	6	7	8	9	10	11
`C`	Lyndon	**0.02**	**0.03**	**0.05**	**0.14**	**0.52**	1.64	4.90	29.21
`C`	standard	**0.02**	**0.03**	0.05	0.15	0.55	1.74	5.31	32.34
`O`	Lyndon	0.05	0.14	0.34	1.21	3.26	10.18	30.46	95.98
`O`	standard	0.05	0.15	0.36	1.25	3.38	10.80	32.89	107.35
`S`	Lyndon	0.06	0.12	0.20	0.37	0.70	**1.41**	**2.87**	**6.01**
`S`	standard	0.06	0.11	0.21	0.37	0.70	1.43	2.92	6.15
`esig`	standard	3.71	8.52	19.26	43.66	98.47	224.83	506.25	1132.24

Table 4.4: Various log signature calculation timings in seconds, for 1000 random 2-dimensional paths of 100 steps each for the given levels

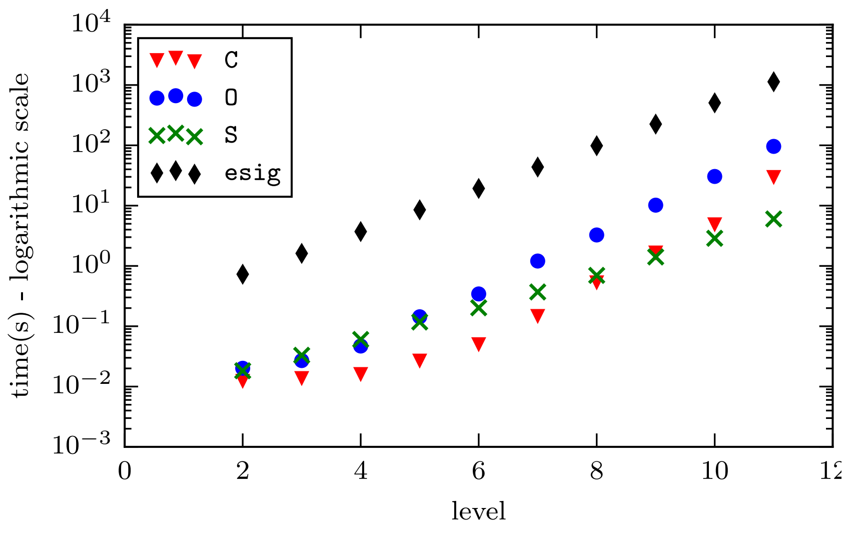


Figure 4.3: Various log signature calculation timings in seconds, for 1000 random 2-dimensional paths of 100 steps each for various levels. For `iisignature`, only the Lyndon basis is shown.

	level:	4	5	6	7	8	9	10
`C`	Lyndon	**0.01**	**0.02**	**0.08**	**0.45**	4.35	40.80	221.41
`C`	standard	0.01	0.02	0.09	0.50	5.36	47.11	255.75
`O`	Lyndon	0.03	0.11	0.51	2.76	14.67	84.04	466.12
`O`	standard	0.02	0.12	0.53	2.98	16.50	101.06	605.05
`S`	Lyndon	0.03	0.05	0.15	0.46	**1.52**	**5.38**	**19.25**
`S`	standard	0.03	0.05	0.15	0.51	1.92	8.36	41.06
`esig`	standard	1.54	5.59	22.69	86.24	338.08	1310.86	5451.14

Table 4.5: Various log signature calculation timings in seconds, for 1000 random 3-dimensional paths of 10 steps each for the given levels

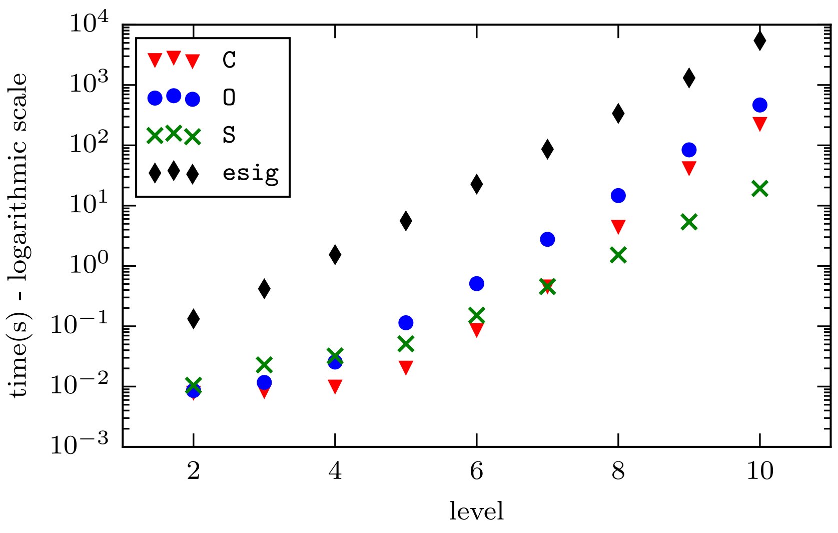


Figure 4.4: Various log signature calculation timings in seconds, for 1000 random 3-dimensional paths of 10 steps each for various levels. For `iisignature`, only the Lyndon basis is shown.

We expect the time taken to increase polynomially in dimension, so we plot the time taken for various methods as d increases on a log-log plot in Figure 4.5 and show the timings in Table 4.6. Ultimately the calculation time would be expected to be quintic in d. For d in the high single figures, we observe much higher growth in the runtime of the compiled code.

	dimension:	4	5	6	7	8	9	10
`C`	Lyndon	**0.09**	**0.29**	0.85	3.17	7.24	22.33	37.83
`C`	standard	0.10	0.30	1.03	3.35	7.58	21.64	38.38
`O`	Lyndon	0.51	1.63	4.18	9.34	20.76	39.39	63.67
`O`	standard	0.52	1.69	4.38	9.84	19.72	38.44	64.92
`S`	Lyndon	0.14	0.37	**0.84**	**1.73**	**3.24**	**5.66**	**9.30**
`S`	standard	0.15	0.41	0.96	2.02	4.56	6.81	11.60
`esig`	standard	22.75	73.17	190.47	437.35	931.58	1814.74	

Table 4.6: Various level-5 log signature calculation timings in seconds, for 1000 random paths of 10 steps each of various dimensions.

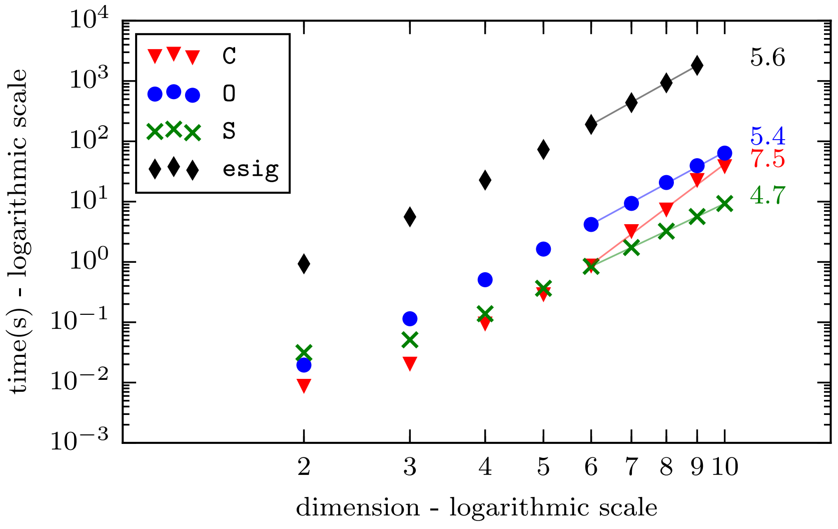


Figure 4.5: Various level-5 log signature calculation timings in seconds, for 1000 random paths of 10 steps each of various dimensions. For `iisignature`, only the Lyndon basis is shown. The graphs look to have roughly reached a straight line for $d \geq 6$. The least squares line of each is shown, with its gradient which indicates the approximate degree of a polynomial relationship.

The preparation in `iisignature` is slow for the `C` method when d or m is large. Timings for a single call are illustrated for various levels with $d = 3$ in Figure 4.6 and for various dimensions with $m = 5$ in Figure 4.7. There is an advantage in using the Lyndon basis.

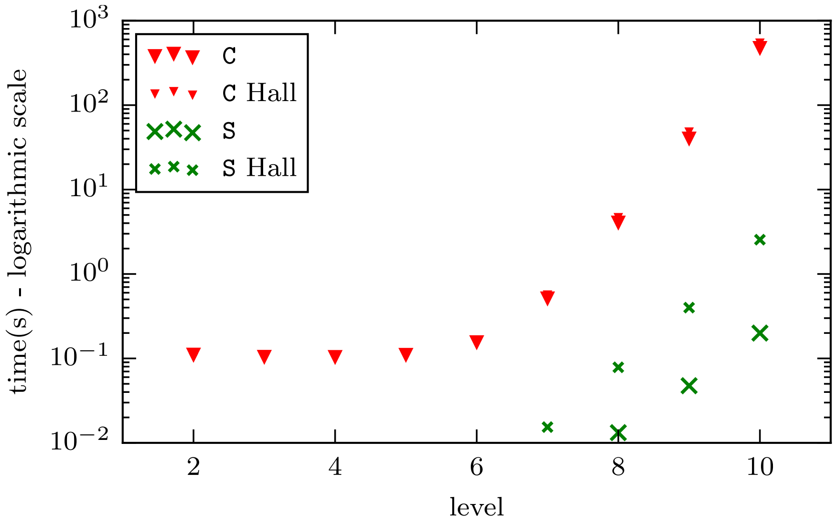


Figure 4.6: Timings for a single preparation of the 3-dimensional log signature calculation for various levels. Smaller marks are used for the standard Hall basis, regular marks for the Lyndon basis. Values for `O` and `C` are very similar, so the former are omitted. Very small values are also omitted.

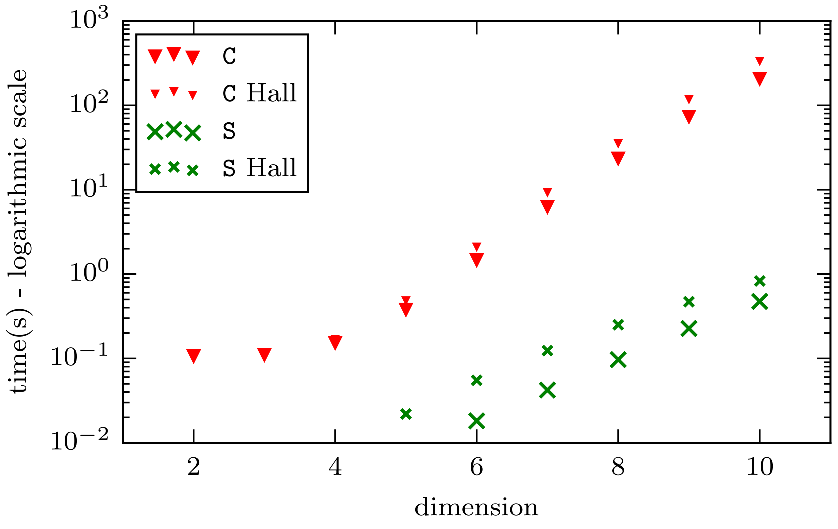


Figure 4.7: Timings for a single preparation of the level-5 log signature calculation for various dimensions. Only the Lyndon basis is shown. Smaller marks are used for the standard Hall basis, regular marks for the Lyndon basis. Values for `O` and `C` are very similar, so the former are omitted. Very small values are also omitted.

While there is always more that can be done to speed software performance, `iisignature` provides a significant speedup over other options easily available to those using python for machine learning and doing lots of signature calculations.

4.7 Indicative memory usage

We used the Massif tool [NWF06] from Valgrind to profile memory usage calculating a single log signature, in particular collecting the peak memory usage. The peak memory usage is interesting because running out of memory is usually what makes certain calculations impossible. There is a background memory cost independent of the algorithm, which includes space to store the BCH coefficients. In order just to measure the algorithm, we ran these calculations from within C++ without using Python, and we subtract the memory usage calculating the same signature at level 1 from the observed memory usage.

The values are very consistent across repeated runs. Values for three-dimensional paths are shown in Figure 4.8. We observe that memory usage is another reason why the projection method becomes a better choice for higher levels.

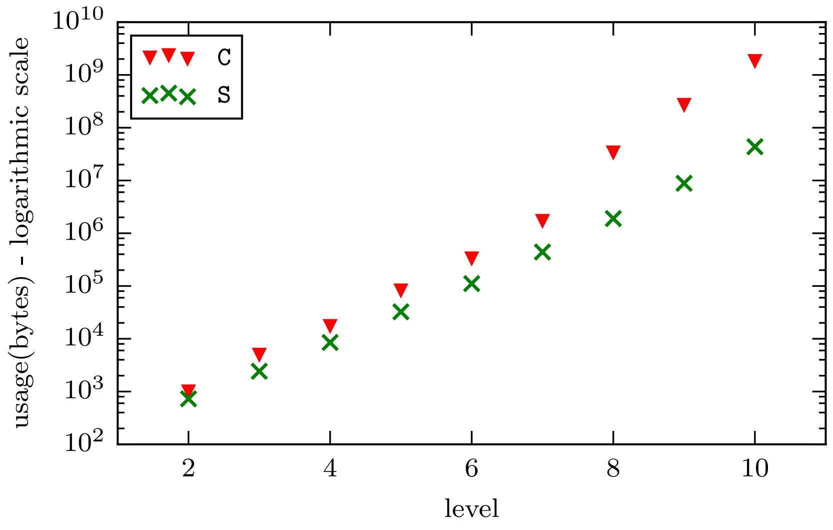


Figure 4.8: Memory usages in bytes for `C` and `S` calculations of a three-dimensional path of 10 steps for various levels. Only the Lyndon basis is shown.

4.8 Conclusions and future work on iisignature

We have presented and analysed efficient methods for computing signatures and log signatures. We have implemented what we considered to be the most promising algorithms.

We have focused on small-dimensional data in our design, because this encompasses many applications where the signature has been used in machine-learning. For example handwriting recognition and EEG data. Calculating the log signature in cases where $d \gg m$ has not been a priority. For example $d > 50$ and $m \leq 4$. Some data is naturally a high-dimensional sequence, possibly discrete (making movements of a fixed length in a single dimension at a time), like some representations of music and text, so this is a possible use case. In these cases there is lots of repetition in the calculations. There are potential changes to the code which would make `prepare` use significantly less memory (and therefore be usable for larger d) in this regime, at the cost of a little more calculation time in `logsig`.

We have shown that machine code generation directly from the algebra is useful in this domain. The calculation is data-access heavy and the order of operations has a big effect, because of memory latency and data dependencies, and there is scope to improve it. We find that adding extra operations to the code without changing the data access does not slow it down, suggesting that it is the effect of data-cache misses which is the main bottleneck. There are subsets of the calculation which are repeated on different parts of the data. Operating in parallel with vector instructions might speed things up. There are avenues for working on these possibilities, for example using the LLVM system, which brings the advantages of a modern compiler to the code generation.

4.9 Backpropagation and derivatives

When training neural networks, it is required to calculate the gradient of a scalar function of many variables. The context for this will be fleshed out in subsection 5.3.2. This is done using the chain rule in a structured fashion, known as *reverse automatic differentiation* or *backpropagation*. In order to be able to do signature calculations inside neural networks, we need to do backpropagation of derivatives through the calculation. The key for this is that for each calculation which takes a number of inputs to a number of outputs, we need to be able to take the derivatives of some scalar function, say F, of the outputs to the derivatives of that function of the inputs. Explicitly, if as a general component of a neural network we wish to have a function[2]

$$f : \mathbb{R}^n \to \mathbb{R}^m$$
$$f : (x_1, \ldots, x_n) \mapsto (f_1, \ldots, f_m)$$

then we need to calculate a backpropagation operation

$$\begin{aligned}\overleftarrow{f} : \Big(\frac{\partial F}{\partial f_1}, \ldots, \frac{\partial F}{\partial f_m}\Big) &\mapsto \Big(\frac{\partial F}{\partial x_1}, \ldots, \frac{\partial F}{\partial x_n}\Big) \\ &= \Big(\sum_i \frac{\partial F}{\partial f_i}\frac{\partial f_i}{\partial x_1}, \ldots, \sum_i \frac{\partial F}{\partial f_i}\frac{\partial f_i}{\partial x_n}\Big)\end{aligned}$$

where all partial derivatives are evaluated at a single set of inputs to the whole function F, and these inputs and the corresponding $(x_1, \ldots, x_n)$ and $(f_1, \ldots, f_m)$ can be made inputs to $\overleftarrow{f}$. In this section I discuss ways I have done that efficiently for some functions in `iisignature`. In particular, for many functions f in practice it is not necessary to calculate or store every element $\frac{\partial f_i}{\partial x_j}$ of the Jacobian. `iisignature` has a simple method `sigjoin` for concatenating a segment onto a signature, and `sigscale` for transforming a given signature according to the path being scaled by an enlargement in each of the coordinate directions. It is these and the core `sig` and `logsig` functions which `iisignature` provides backpropagation for.

4.9.1 General thoughts

Having a backpropagation function along with a library function can often be more efficient than defining the whole operation in the language of an autodifferentiation facility like those in `tensorflow`, `torch` or `theano`. There are lots of steps in signa-

[2] A function may take scalar inputs or other inputs with respect to which we do not need derivatives, in which case these extra inputs can be inputs to its backpropagation operation and no special extra difficulty is caused.

ture calculations which would each have to be analysed separately, and this process would probably happen every time a program was run. Each calculation we provide together with its backprop counterpart can be thought of by the deep learning library as a sealed unit. Often we can save memory this way – we don't have to store every single intermediate value, and at runtime we don't need to work out which intermediate values need to be stored. Our function can be provided to an autodifferentiation facility as a sealed unit.

The most common style is for a backprop function to receive the original function's inputs. We have functions which have optional positional arguments. Therefore it is most consistent to take the gradient as the first argument of backprop functions followed by the original inputs. In practice, a backprop function could additionally accept original function's output and the autodifferentiation systems would be able to provide it, but we don't have a need to make use of this. Ideally state which is internal to the calculation should not need to be remembered.

In some cases, by thinking carefully about the backpropagation operation $\overleftarrow{f}$ we can find a more sensible way to calculate it than the obvious way. This means that our own function can be more efficient than that which an autodifferentiation system is likely to have come up with. In practice, it seems that our $\overleftarrow{f}$ pleasingly take about the same time as the corresponding forwards operation f, and it would be unlikely that the backpropagation operation would be much quicker in any case.

When we have a calculation structure which looks like Figure 4.9 to backpropagate derivatives through, we need the value of the output of each calculation of function f as we pass derivatives down the tree, in order to send the derivative to the input which is a sibling of that f. There are basically two ways to make this available. The standard way is that we evaluate the calculation forwards storing all the intermediate f values. This may require extra memory and copying of data to store them (unless we repeatedly do parts of the calculation). This is classic reverse-mode automatic differentiation in action, and is what typical deep learning packages do by default to implement backpropagation. Some of the storage can be omitted in return for redoing parts of the calculation. In some cases, however, we might be lucky, in that both f and (right multiplication by something fixed) are invertible, and so we can find the output of each f down the tree starting from the output of the calculation. By doing this in tandem with the backpropagation of derivatives, we don't need any storage for each level of the calculation graph. This idea works for `sig` but not for converting a signature to a log signature using the series for log, fundamentally because among group-like elements of tensor space, multiplication by a given element is invertible, but among Lie elements, multiplication by a given element is not.

$$\begin{tikzcd}
\text{input} \arrow[r] & \times \arrow[r] & f \arrow[r] & \times \arrow[r] & f \arrow[r] & \times \arrow[r] & f \arrow[r] & \times \arrow[r] & \text{output} \\
\text{input} \arrow[ur] & & \text{input} \arrow[ur] & & \text{input} \arrow[ur] & & \text{input} \arrow[ur] & &
\end{tikzcd}$$

Figure 4.9: A calculation dependency, where f is some function and $\times$ is something like multiplication.

4.9.2 sig

There is nothing so new here, but it helps to write this function's calculation out to fix notation for when we come to the derivative.

`sig(A,m)` calculates the signature of a piecewise linear path by combining the signatures of each straight-line-piece (which are the exponents of the displacement) using Chen's formula.

If the d-dimensional input was $\left((p_{jk})_{j=0}^{l}\right)_{k=1}^{d}$ I could express the calculation like this. The displacement vectors are given by

$$(q_j)_k = p_{jk} - p_{(j-1),k} \tag{4.6}$$

and the signature of the j segment is

$$r_j(i_1 \dots i_n) = \frac{1}{n!}\prod_{h=1}^{n}(q_j)_{i_h} \tag{4.7}$$

and using Chen's relation we find the signature of the path up to point j

$$\begin{aligned} s_1(i_1 \dots i_n) &= r_1(i_1 \dots i_n) \\ s_j(i_1 \dots i_n) &= s_{j-1}(i_1 \dots i_n) + \left[\sum_{p=1}^{n-1} s_{j-1}(i_1 \dots i_p) r_j(i_{p+1} \dots i_n)\right] + r_j(i_1 \dots i_n). \end{aligned} \tag{4.8}$$

I adopt the convention that a nonsensical ellipsis like $i_1 \dots i_0$ means the empty word ϵ. The value of a signature on the empty word is the scalar 1. This can therefore be written simply

$$s_j(i_1 \dots i_n) = \sum_{p=0}^{n} s_{j-1}(i_1 \dots i_p) r_j(i_{p+1} \dots i_n).$$

Each signature object is calculated all at once (i.e. its value for all words is calculated in one go, for all words up to length m), and stored in a vector (one for each level/length of word) of vectors (of words in alphabetical order). The output of the function is all the values of s_l. For example, for a path given as six points, the calculation dependency looks like Figure 4.10, taking the displacement vectors and signature objects as single entities and neglecting the actual calculation of the displacements. The calculation order needn't be like this, the tree could be constructed differently, but the number of operations would be unchanged. A nested calculation order like this requires only storing one s_j at a time.

$$\begin{tikzcd}
 & & & & s_5 & \\
 & & & s_4 \arrow[ur] & & r_5 \arrow[ul] \\
 & & s_3 \arrow[ur] & & r_4 \arrow[ul] & q_5 \arrow[u] \\
 & s_2 \arrow[ur] & & r_3 \arrow[ul] & q_4 \arrow[u] & \\
s_1 = r_1 \arrow[ur] & & r_2 \arrow[ul] & q_3 \arrow[u] & & \\
q_1 \arrow[u] & & q_2 \arrow[u] & & &
\end{tikzcd}$$

Figure 4.10: Sig calculation dependency

I use the $\underline{\otimes}$ symbol for the Kronecker product of two column vectors as a vector, which takes a vector of length x and a vector of length y to a vector of length xy. If u has length x and v has length y then uv has elements $(u\underline{\otimes}v)_{(i-1)y+j} = u_i v_j$. If a is a signature, then I write a^m for the vector of its elements at level m. To calculate r_j I use the procedure in Algorithm 1 and the algorithm for the full signature is Algorithm 2.

Algorithm 1 Segment signature

```
1: $r_j^1 \leftarrow q_j$
2: for $n \leftarrow 2, m$ do
3:     $r_j^n \leftarrow \frac{1}{n}(r_j^{n-1} \underline{\otimes} q_j)$
4: end for
```

It feels wasteful to store each r_j^n as a whole vector of d^n numbers, because they are really repeats. The signature of a straight line takes the same value on all permutations of a word, so there are only $\binom{d+n-1}{n}$ distinct values. But trying to exploit this led me to slower, more complicated code.

Algorithm 2 sig

```
 1: Calculate $r_1$
 2: for $n \leftarrow 1, m$ do
 3:     $s^n \leftarrow r_1^n$
 4: end for
 5: for $j \leftarrow 2, l$ do
 6:     Calculate $r_j$
 7:     for $n \leftarrow m, 1$ do                    ▷ make $s^n$ go from being $s_{j-1}^n$ to $s_j^n$
 8:         for $n' \leftarrow (n-1), 1$ do
 9:             $s^n \leftarrow s^n + s^{n'} \underline{\otimes} r_j^{n-n'}$
10:         end for
11:         $s^n \leftarrow s^n + r_j^n$
12:     end for
13: end for
14: $s^n$ is now $s_l^n$ for each $n$
```

4.9.3 sigbackprop

Naively, given a calculation tree to define an output in terms of simple calculations, we can backpropagate derivatives through the tree using the chain rule. If we know the value of every node in the tree (having stored them while calculating the output) then we can pass a derivative though each leg of a multiplication node M by multiplying it by the stored values in the other inputs to M. We send a derivative both ways through an addition unchanged. There is naturally a time/memory tradeoff in such a calculation, because each value could just be recalculated from inputs when it is needed, instead of having been remembered. We can in fact do much better than storing all the intermediate values, or even just all the s_j.

In general, if X is the signature of a path γ, then the signature X' of the reversed path γ^{-1} is a permutation with some elements negated. It is the image of X under α, the antipode of the concatenation algebra (page 19 of [Reu94]).

$$X'(a_1 \dots a_n) = (-1)^n X(a_n \dots a_1) \tag{4.9}$$

If γ is a straight line, then the component of the signature is the same on all permutations of a word, and so we have the simpler

$$X'(a_1 \dots a_n) = (-1)^n X(a_1 \dots a_n) \tag{4.10}$$

Because all the r_j are easy to calculate as the difference of inputs, we can easily use this to calculate s_{j-1} from s_j. This we can perform at the same time as the backpropagation of derivatives with respect to s_j. Let the vector containing level n

of r_j be r_j^n, with the derivatives with respect to its elements being R_j^n. Similarly s_j^n and S_j^n for s_j.

It is convenient to define the corresponding backpropagation operations to $\underline{\otimes}$. If u has length x, v has length y, and w has length xy, then $w \underset{\leftarrow}{\otimes} v$ has length x with elements $(w \underset{\leftarrow}{\otimes} v)_i = \sum_j w_{(i-1)y+j} v_j$ and $u \underset{\rightarrow}{\otimes} w$ has length y with elements $(u \underset{\rightarrow}{\otimes} w)_j = \sum_i u_i w_{(i-1)y+j}$. Then the algorithm, based in the function `sigBackwards`, proceeds as shown in Algorithm 3. Some of the same logic is used for `sigjoinbackprop`.

Algorithm 3 sigBackwards

```
 1: for $n \leftarrow 1, m$ do
 2:     $s^n \leftarrow s_l^n$, calculated using sig
 3:     $S^n \leftarrow S_l^n$, an input
 4: end for
 5: for $j \leftarrow l, 1$ do
        ($s^n$ is now $s_j^n$ and $S^n$ is now $S_j^n$ for each $n$.)
 6:     Calculate $r_j$.
 7:     Use (4.10) to make $s^n$ be $s_{j-1}^n$ for each $n$.
            for $n \leftarrow m, 1$ do
                for $n' \leftarrow (n-1), 1$ do
                    $s^n \leftarrow s^n + (-1)^{n-n'} s^{n'} \underline{\otimes} r_j^{n-n'}$
                end for
                $s^n \leftarrow s^n + (-1)^n r_j^n$
            end for
 8:     Calculate $R_j$.
            for $n \leftarrow 1, m$ do
                $R_j^n \leftarrow S^n$
            end for
            for $n \leftarrow 1, m$ do
                for $n' \leftarrow (n-1), 1$ do
                    $R_j^{n-n'} \leftarrow R_j^{n-n'} + s^{n'} \underset{\rightarrow}{\otimes} S^n$
                end for
            end for
 9:     Backpropagate $R_j^n$ through (4.7).
10:     Make $S^n$ be $S_{j-1}^n$, doable in place.
            for $n' \leftarrow 1, (m-1)$ do
                for $n \leftarrow (n'+1, m)$ do
                    $S^{n'} \leftarrow S^{n'} + S^n \underset{\leftarrow}{\otimes} r^{n-n'}$
                end for
            end for
11: end for
```

4.9.4 sigscalebackprop

Here we present a general sensible idea to use when differentiating products of terms sampled with replacement from a list: *you keep the simple code you would have written if replacement was not allowed.*

Consider `sigscalebackprop` called in d dimensions up to level m, in particular its action for a single level $l \leq m$. Each level's calculation produces derivatives with respect to its part of the original signature, and contributes to derivatives with respect to the scales. The data are a, the lth level of the original signature, b, the scales, and E, the supplied derivatives with respect to the lth level of the output. Denote the output of the expression by e and the to-be-calculated derivatives with respect to a and b by A and B respectively. I denote subscripting of the inputs and outputs with square brackets, and unlike earlier where I indexed a level of a signature as a vector, here I am indexing it as a tensor.

The expression to be differentiated is as follows.

$$e[i_1,\dots,i_l] = \Big(\prod_{j=1}^{l} b[i_j]\Big) a[i_1,\dots,i_l] \qquad \text{for} \quad (i_1,\dots,i_l) \in \{1,\dots,d\}^l \tag{4.11}$$

Using standard rules for differentiation, the derivative calculations are as follows.

$$A[i_1,\dots,i_l] = \Big(\prod_{j=1}^{l} b[i_j]\Big) E[i_1,\dots,i_l] \qquad \text{for} \quad (i_1,\dots,i_l) \in \{1,\dots,d\}^l \tag{4.12}$$

$$B[k] = \sum_{\substack{(i_1,\dots,i_l)\in\\ \{1,\dots,d\}^l}} B[k;i_1,\dots,i_l] \qquad \text{for} \quad k \in \{1,\dots,d\}, \tag{4.13}$$

where the single contribution of a product is

$$B[k;i_1,\dots,i_l] = \Big(\prod_{\substack{j=1\\ i_j\neq k}}^{l} b[i_j]\Big) a[i_1,\dots,i_l] E[i_1,\dots,i_l]\ C^k_{i_1\dots i_l} b[k]^{C^k_{i_1\dots i_l}-1} \tag{4.14}$$

and $C^k_{i_1\dots i_l}$ is the number of j for which $i_j = k$.

Evaluating e according to (4.11) will take time $d^l l$ but evaluating B naively according to this procedure takes at least $d^{l+1}d$, because evaluating the d counts $C^{\cdot}_{i_1\dots i_l}$ requires d time for every $(i_1 \dots i_l)$, even though most are zero. A couple of rearrangements:

$$B[k;i_1,\dots,i_l] = \Big(\prod_{j=1}^{l} b[i_j]\Big) a[i_1,\dots,i_l] E[i_1,\dots,i_l]\ C^k_{i_1\dots i_l} b[k]^{-1} \tag{4.15}$$

$$B[k;i_1,\dots,i_l] = \sum_{h=1}^{l} 1_{\{i_h=k\}} \Big(\prod_{j=1}^{l} b[i_j]\Big) a[i_1,\dots,i_l] E[i_1,\dots,i_l]\ b[i_h]^{-1} \tag{4.16}$$

Organising the terms differently by making k range among $(i_1, \dots, i_l)$ instead of $\{1, \dots, d\}$ means we split the $C^{\cdot}_{\cdot}$ occurrences of each multiplication. This leads to Algorithm 4, which only takes time $d^l l$.

Algorithm 4 single level of sigscalebackprop

```
1: for $(i_1, \dots, i_l) \in \{1, \dots, d\}^l$ do
2:     $prod \leftarrow \prod_{j=1}^{l} b[i_j]$
3:     $A[i_1, \dots, i_l] \leftarrow prod\; E[i_1, \dots, i_l]$
4:     for $h \in \{1, \dots, l\}$ do
5:         $B[i_h] \leftarrow B[i_h] + prod\; a[i_1, \dots, i_l] E[i_1, \dots, i_l] / b[i_h]$
6:     end for
7: end for
```

Chapter 5

Signatures in deep learning

5.1 Introduction

The Chinese and Japanese languages are very popular and have similar writing systems. With the widespread use of smartphones and touch devices in the last decade, automated recognition is relevant to allow text input via letter drawing rather than by keyboard, which can be inefficient. The leading mobile operating systems, Apple's iOS and Google's Android, provide built-in handwritten input, and third-party apps are also available. Mobile devices have limited compute power because of battery life. Therefore computationally efficient methods are important.

There is a long history of methods for handwritten Chinese character recognition. Much of the work has treated the case of recognising the input from an image, for example of a paper manuscript. This is termed *offline* character recognition, and is distinguished from *online* character recognition where the data is given as a sequence of strokes with their points supplied in order. The order and directions of the strokes is potentially useful, so better accuracy is potentially possible in the online setting by exploiting the extra information available. The online problem is the relevant one for handwriting interfaces on devices.

When I started looking at this problem in 2014, there were a number of methods for performing online Chinese recognition, each with some pros and cons. The first promising method used an 8x8 spatial grid of 8-directional histograms [BH05]. The approach showed the importance of capturing both the location of the pen, and the direction of the pen's movements. The method is computationally fairly cheap, but the accuracy was limited. Ad hoc pre-processing was needed to reshape the characters to deal with the lack of awareness of translation invariance of human handwriting. Given a circle, there is a large margin of error if you try to calculate the radius from the 8x8x8 histogram due to the coarse resolution of the 8x8 spatial grid.

From around 2010, the most successful methods were using large convolutional neural networks. They capture the identity and shape of each stroke separately, together with order. The winner of the Chinese Handwriting Recognition Contest 2010 [Liu+10] with 92.39% accuracy (on the same data as we use) used histogram features along with such a large network.

Schmidhuber and collaborators [CMS12] used a higher resolution convolutional network which was therefore more computationally expensive, achieving greater accuracy (94.39%). This was not using signatures

Combining convolutional networks with signatures boosted accuracy but the computational cost was still rather high – 96.18% accuracy by Ben Graham in [Gra13] was the best in the ICDAR 2013 Chinese Handwriting Recognition Competition [Yin+13].

One of the leading third-party apps is GPEN [Jin] from the South China University of Technology, one of their papers from a bit later is [Yan+15]. They use signature features as part of the representation, and also a large convolutional network. In such apps reducing the required power for classification is always a priority, although efficient training is also attractive.

Recurrent neural networks, such as LSTMs, offer an alternative way to learn sequence data which can be fast compared with convolutional networks. Combining LSTM with signatures was an attractive idea to us as the signatures would capture local shape information potentially very cheaply, allowing the deep learning problem of combining the partial signatures to be relatively small, and so we wanted to investigate this possibility. We constrain ourselves to working with the raw data, as we wish to develop algorithms that can generalise well, rather than needing human-intensive problem specific work on feature engineering which requires much trial and error. The test set only has 60 writers, and much of the test error is concentrated on a few of them. There is a danger that feature engineering results in methods that work well on the test set, but does not generalise well to writers outside the dataset.

An example of LSTM with feature engineering and customised data augmentation is [Zha+18] which first appeared on ArXiv in June 2016 and then showed the best accuracy yet, albeit using an enlarged training set. It shows LSTM is a good fit for the problem domain, but powerful universal local shape characterisation is desirable.

Here, I will introduce some background to the problem and deep learning methods in general, and show a potential pipeline for using signatures in combination with recurrent neural networks.

5.2 Description of data

The CASIA datasets [Liu+11] are large standard datasets for Chinese handwriting recognition. The task we are attempting is the online character recognition problem. We pick one of the standard databases for the task, namely the online handwritten character database 1.1 (`OLHWDB1.1`).[1] There are samples of 3755 different characters (both symbols and Chinese characters) written by 300 different writers with a special *Anoto* pen which records each stroke as a sequence of coordinates. This data is

[1] Another suggested procedure, for example used in [Zha+18], is to combine data from both the 1.0 and 1.1 datasets. This is actually recommended in [Liu+11].

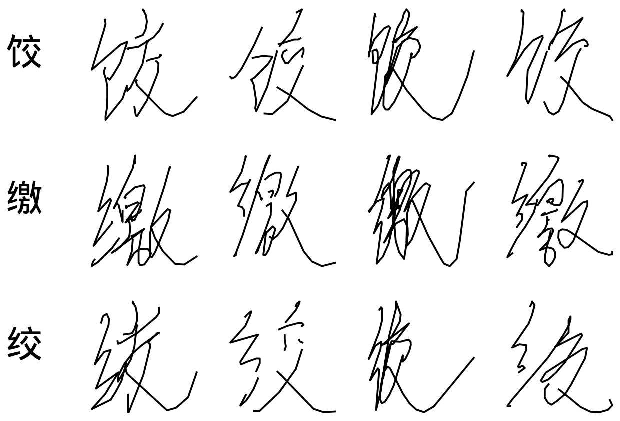


Figure 5.1: Three Chinese characters and their handwritten versions as provided by each of the first four writers in the CASIA `OLHWDB1.1` training set. Note that the number of strokes in a single character can vary between writers. (These are Unicode code points `0x997a`, `0x7f34` and `0x7ede`.)

directly comparable to data from a touch interface. We split them in the standard recommended train-test split, so that the first 240 writers' data is used for training and the last 60 for testing. The data in total consists of 939,564 labelled sample characters for training and 234,800 for testing. The order of writers is random, so both the training set and the testing set should be representative of the pool of writers. (This is not true in the 1.0 database). On average, a character has 5.6 strokes, with the maximum number in a character being 26. On average, a character is described by a total of 60.9 points, with the maximum being 283. A stroke is described by on average 10.9 points, with the maximum being 192. Figure 5.1 shows some examples of three characters from the training set. A Chinese native described the handwriting to me as bad but readable.

5.3 Generic supervised learning setup

The typical setup in supervised machine learning is that we want to learn an unknown function f from training example space X to output space Y. We are given a *training set* of examples from X for which $f(X)$ is known.

5.3.1 Augmenting the output space

Often the output of a network needs to be a probability distribution rather than discrete values, and we define a loss function to optimise. We define a possibly augmented output space $\hat{Y}$ which is continuous and for which there is a simple deterministic function $i : \hat{Y} \twoheadrightarrow Y$. We also define a cost function $c : Y \times \hat{Y} \to \mathbb{R}_0^+$ where a high value of $c(y, \hat{y})$ indicates that the prediction $\hat{y} \in \hat{Y}$ is bad when the right answer is y.

We make some form of a model, a function $\tilde{f}$, which takes a sample $x \in X$ and set of parameters λ in parameter space Λ to the constructed output space $\hat{Y}$. The aim of training is to come up with a $\lambda \in \Lambda$ such that for a sample x, $i(\tilde{f}(x, \lambda))$ is roughly $f(x)$.

We pick $\hat{Y}$, i, and c in conjunction with the model so that changing λ to reduce $c(f(x), \tilde{f}(x, \lambda))$ makes $i(\tilde{f}(x, \lambda))$ a good approximation of $f(x)$ and also so that $c(f(x), \tilde{f}(x, \lambda))$ is a differentiable function of λ. In practice, $\tilde{f}$ is not only differentiable but its derivative is also easy to calculate, via the chain rule, in the form of autodifferentiation and the method of backpropagation of derivatives, as mentioned in section 4.9. If our formalism didn't allow for $\hat{Y} \neq Y$ then we would have a problem with allowing our cost to be differentiable with respect to the parameters if Y was a discrete space.

For example, if you were training a model to distinguish pictures of cats from pictures of dogs, Y would naturally be the set $\{\text{dog}, \text{cat}\}$. In this case a sensible choice would be as follows.

$$\begin{aligned}
\hat{Y} &= [0, 1] \\
i(\hat{y}) &= \begin{cases} \text{dog} & \hat{y} \leq 0.5 \\ \text{cat} & \hat{y} > 0.5 \end{cases} \\
c(y, \hat{y}) &= \begin{cases} -\log \hat{y} & y = \text{dog} \\ -\log(1 - \hat{y}) & y = \text{cat} \end{cases}
\end{aligned}$$

The intuition here is that we make an element of augmented output space be a probability that a picture is a cat. The amount by which we are wrong, our cost, is the cross entropy between the true answer and the predicted answer.

If there are $k > 2$ categories, we might choose the following, a one-hot encoding of the output:

$$\begin{aligned}
Y &= \{1, \ldots, k\} \qquad \text{say,} \\
\hat{Y} &= \mathbb{R}^k \\
i(\hat{y}) &= \operatorname*{arg\,max}_{m \in \{1, \ldots, k\}} (\hat{y})_m \\
c(y, \hat{y}) &= -\log(\operatorname{softmax}(\hat{y})_y)
\end{aligned}$$

where softmax : $\hat{Y} \to \hat{Y}$ given by

$$\operatorname{softmax}(\hat{y})_m = \frac{\exp(\hat{y}_m)}{\sum_{m' \in \{1,\ldots,k\}} \exp(\hat{y}_{m'})}$$

is a smooth function which maps vectors to the standard simplex Δ^{k-1}, i.e. discrete probability distributions. c is the cross-entropy of this distribution, or the negative-log likelihood of the model on the data.

As another example, we might be trying to learn a function whose image is $[0,1]$. In this case, we have no need for augmentation and a sensible choice might be as follows:

$$\begin{aligned} \hat{Y} &= Y = [0,1] \\ i(\hat{y}) &= \hat{y} \\ c(y,\hat{y}) &= (\hat{y} - y)^2 \end{aligned}$$

The cost function is the square of the L^2 norm.

5.3.2 Minibatches and stochastic gradient descent

With this setup, we can formulate training as a problem of stochastic gradient descent. On a subset of n of the training examples $\chi \in X^n$, the badness of some set of parameters λ is given by

$$g_\chi(\lambda) = \sum_{x \in \chi} c(f(x), \tilde{f}(x,\lambda)). \tag{5.1}$$

One way to find the best parameters would be to set χ to the entire training set and numerically minimise $g_\chi(\lambda)$. We have first derivatives of g_χ so gradient descent would be a possibility. But g_χ would be very slow to calculate, as it would require passing over the whole dataset, and this would have to happen very many times.

We can think of $g_\chi(\lambda)$ as a random function as χ is uniformly distributed over some samples from the training data. It is much more efficient to try to minimize this random function as a stochastic gradient descent. The typical procedure is to start by picking once some initial random parameters λ, then to repeat the following many times: pick a random subset χ of the training data, evaluate $g_\chi(\lambda)$ and $g'_\chi(\lambda)$, and update λ using some "stochastic gradient descent" technique designed to minimise $g_\chi(\lambda)$.

The dimensionality of λ can be large. Typically $\Lambda = \mathbb{R}^d$ for d in the millions or billions. It is generally not feasible to calculate or even store the second derivative $g''_\chi(\lambda)$ because it has d^2 elements. Thus we cannot use minimisation techniques like Gauss-Newton or Levenberg-Marquardt which use the second derivative to determine the length of a step. $g'_\chi(\lambda)$ can give us an idea of a direction to move λ, but not an amount. A simple minimisation technique which can work is to move a fixed amount (called the *learning rate*) in a direction given by an exponentially weighted moving average of recently seen values of $-g'_\chi(\lambda)$. This is called *momentum*. There are a few other techniques which work well. They have in common that the memory usage grows no more than linearly in d and not at all in the number of steps of optimisation history available.

5.3.3 Inductive bias

When choosing a supervised machine learning model, we are choosing what kinds of functions are relevant in trying to find one which agrees with our training data. The *inductive bias* is the set of built-in assumptions about functions to be considered which our choice implies.

5.3.4 Invariants

Across supervised machine learning having more training examples enables building better models.

If there are known symmetries, or approximate symmetries, in the domain of the function being learned, then it can make learning much more effective to take this into account. Only certain functions have any chance of being useful approximations of the target function f, those which are generally unaffected by the symmetry. If we have a true group of symmetries, where we know $f(s(x)) = f(x)$ for all s in some group of transformations, then we want f to take a single value on each orbit. Often the most effective way to take a symmetry into account is to restrict our model to a form of function which has (approximately) the desired symmetry, or which makes it easy to learn functions which have the desired symmetry. For example, when learning a function on images we may use a convolutional neural network which often will return similar values under small translations of the input. Similarly, when using a 1D convolutional neural network to classify a time series which I thought to be translation invariant (for example EEG voltage readings) I restricted the first convolutional layer to have multiplicative weights which sum to 0. Significantly, the fact that the signature is invariant under reparametrisations of a path leads to its possible use as a source of features of paths in the common case that we know that reparametrisation should not matter.

5.3.5 Data augmentation

Often some symmetry or invariant in the domain cannot be built into the structure of the model, because it would be too complicated. But the invariant still means that a single piece of labelled training data can count as more than one example, because some distortions of it is expected to be also a good training example. Data augmentation in this way is an important part of making the most of a dataset. For a given problem, we might establish a set of transforms under which our data would still be good, including the identity transform. When generating each training sample in each χ, we put it through a randomly chosen transform.

For example, if you were training a simple convolutional neural network to distinguish pictures of cats from pictures of dogs, you might exploit the fact that a picture of an animal flipped along a vertical axis is a good synthetic example of a picture of that type of animal. Each training sample might be flipped with probability $\frac{1}{2}$ before use in χ.

There is ongoing research into good data augmentation transformations, for example [Cub+18], and some Python libraries have been made available for images, for example `Augmentor` and `imgaug`. Many of the same transformations used for images can be translated into path data, for example small rotations, shears and elastic distortions, although these libraries are not directly usable in our setting.

5.4 Neural networks

Neural networks (NNs), or artificial neural networks (ANNs), are a category of forms for $\tilde{f}$, our parameterised function which we want to find parameters for to make it solve a supervised learning problem. The function is built from a series of layers, and each layer is made from "cells" or "neurons" in a manner analogous to the brain. *Deep learning* is basically machine learning using neural networks.

5.4.1 Fully connected

The simplest neural networks are *fully connected neural networks* (FCNNs) or *multi-layer perceptrons* (MLPs). A fully connected neural network looks like the following. The input space, X is $\mathbb{R}^{u_0}$ for some u_0. The following must be given: a *number of layers* m, a number of *units* in each layer $(u_i)_{i=1}^{m}$, and a nonlinear function $\sigma : \mathbb{R} \to \mathbb{R}$ called the activation function. The output space is $\hat{Y} = \mathbb{R}^{u_m}$. Then an element λ of

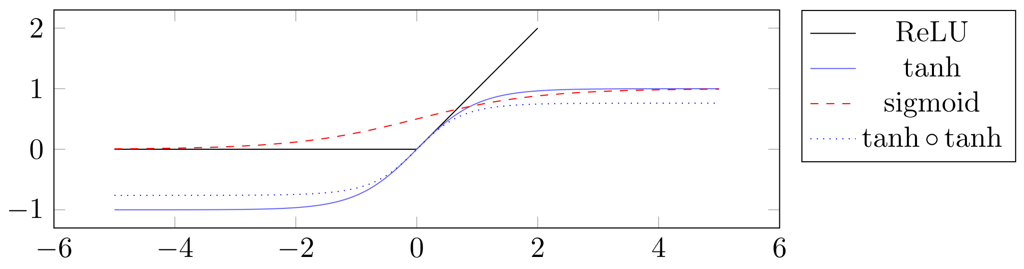


Figure 5.2: Some important activation functions

parameter space is a sequence $(W_i, b_i)_{i=1}^m$ where each W_i is a real $u_{i-1} \times u_i$ matrix called the *weights*, and each $b_i \in \mathbb{R}^{u_i}$ is a vector called the *biases*. If x is a vector, matrix or array, I write $\sigma(x)$ for the result of applying σ to x elementwise.

The neural network is then the function $\tilde{f} = g_n$ defined as follows:

$$g_0(x, \lambda) = x \tag{5.2}$$

$$g_{i+1}(x, \lambda) = \sigma(g_i(x, \lambda)W_i + b_i) \tag{5.3}$$

(There is a commonly encountered equivalent way to formulate this as follows. We have parameters $\lambda = (\tilde{W}_i)_{i=1}^m$ where each $\tilde{W}_i$ is a real $(u_{i-1} + 1) \times u_i$ matrix. Then we define

$$g_{i+1}(x, \lambda) = \sigma\Big(\begin{bmatrix} g_i(x, \lambda) \\ 1 \end{bmatrix} \tilde{W}_i \Big). \qquad) \tag{5.4}$$

There is lots of terminology. For $i > 0$, an element in $g_i(x, \lambda)$ is said to be a *neuron*, a *feature*, a *feature detector*, a *cell* or a *unit* or the activation of one. g_i is the ith *layer*. Anthropomorphically, these values are thought of as the results of intermediate steps the network has learnt to calculate from the input on its way to the output. The structure of a FCNN is unchanged if, as an extra first step, the input is multiplied by a fixed nonsingular matrix or a incremented by a fixed vector. We might say the network "sees" its input as an element of affine space.

The most popular choice for σ is called the *rectifier*, $\sigma(x) = \max(x, 0)$. We say such a network has *rectified linear units* or *ReLUs*, and hence may call σ itself "ReLU". This and some other activation functions which are relevant later are plotted in Figure 5.2.

5.4.2 Convolutional

Fully connected networks take their input to be $X = \mathbb{R}^{u_0}$ with no additional structure. They know of no particular structure of the input space. Often there is some structure which we want to be taken into account. A common case is that we are given data in say $\mathbb{R}^{u_0^0}$ for each point on a lattice of shape $u_0^1 \times u_0^2 \times \cdots \times u_0^d$. Our data lives in the space $X = \mathbb{R}^{\prod_{i=0}^d u_0^i}$. Convolutional neural networks (CNNs) are a generalisation of fully connected neural networks which take this into account. For example, if we are dealing with colour photographs of resolution 28×28 we would have $X = \mathbb{R}^{3\times 28\times 28}$, where we have a value for the brightness of red, green and blue at each pixel. We want functions which treat pixels similarly to each other, and which treat neighbouring pixels similarly, to be favoured by the inductive bias of our model. We want the model to be able to learn a function which doesn't change much if a picture is translated by about one pixel.

A convolutional network allows not just the input but other layers to have a grid structure. The size of the array of values, or the number of units in layer i, $g_i(x, \lambda)$, is usually given by a sequence of numbers $u_i^0 \times u_i^1 \times \cdots \times u_i^d$. u_i^0 is known as the number of *features* in the layer – this is the number of values per point in space. There are several common types of layers. Two of the simplest are the convolutional and max pooling layers. If layer $i + 1$ is an $(l_1 \times \cdots \times l_d)$-*convolutional layer* it contributes a vector $b_{i+1} \in \mathbb{R}^{u_{i+1}}$ and a (typically small) $u_i^0 \times u_{i+1} \times l_1 \times \cdots \times l_d$-array W_{i+1} to Λ. Then, in a simple setup, the elements of the layer's outputs $g_{i+1}(x, \lambda)$ are the values of

$$\sigma\Big(\sum_{j_0=0}^{u_i^0}\sum_{j_1=0}^{l_1}\cdots\sum_{j_d=0}^{l_d}(g_i(x,\lambda))_{(j_0,k_1+j_1,k_2+j_2,\ldots,k_d+j_d)}(W_i)_{(j_0k_0j_1j_2\cdots j_d)} + (b_i)_{k_0}\Big) \tag{5.5}$$

as k_0 ranges from 0 to u_{i+1}^0 and the other k_p range from 0 to $u_{i+1}^p = 1 + u_i^p - l_p$. There are numerous variations on this theme, for example treating the boundary differently or enforcing further sparsity on W_{i+1}. The idea is that the new units at a point in space only depend on the values of the units in the previous layer near that point in space, and the manner of the dependence is the same over all space. This layer is seen as analogous to the V1 cells in a visual cortex. This is equivalent to a fully connected layer where the matrix W is restricted to a very special sparsity pattern, and also that its nonzero values are repeated in a certain way. A max pooling layer takes the maxima of each feature over a each of a grid of small cuboids which either cover each dimension or overlap in each dimension, thus preserving the number of features ($u_{i+1}^0 = u_i^0$) but reducing the other dimensions. It has no parameters. A typical such layer reduces the number of units in each spatial dimension by a factor of two. The highest levels in a typical CNN typically take all the units to be a single vector and thus are fully connected, and so the output of the network happens in the same way as the FCNN.

5.4.3 Recurrent neural networks

The input being a sequence is an important case of a specially-structured input which it is important for the network to take into account. A simple case is that for each example we are given data in say $\mathbb{R}^{u_0^0}$ for each of u_0^1 time points. Our data lives in the space $X = \mathbb{R}^{u_0^0 u_0^1}$. We could be use a 1-D convolutional network (i.e. one with $d = 1$) in this case, but recurrent neural networks (RNNs) are an important alternative type of architecture in this case. While convolutional and pooling layers allow the local spatial structure to be taken into account in the learning, RNNs force the temporal structure into the model, and explicitly globally, not just in a "local" way. An RNN allows the network to allow earlier entries of the sequence to be taken into account when processing later entries. Just like for fully connected and convolutional networks, the network is built from a sequence of layers. The simplest RNN layer could take input as a sequence of u_i^1 values in $\mathbb{R}^{u_i^0}$ to output of the same number ($u_{i+1}^1 = u_i^1$) of values in $\mathbb{R}^{u_{i+1}^0}$ Its parameters would be a weight matrix $u_{i-1}^0 \times u_i^0$ matrix W_i, a second (square) weight matrix $u_i^0 \times u_i^0$ matrix $\bar{W}_i$ for the recurrence, and a u_i^0-vector of biases b_i.

$$(g_{i+1}(x,\lambda))_1 = \sigma((g_i(x,\lambda))_1 W_i + \qquad\qquad b_i) \tag{5.6}$$

$$(g_{i+1}(x,\lambda))_t = \sigma((g_i(x,\lambda))_t W_i + (g_{i+1}(x,\lambda))_{t-1}\bar{W}_i + b_i) \qquad \text{for } t > 1 \tag{5.7}$$

(For a few paragraphs I am omitting the parameters x and λ from g). Intuitively, what is happening here is that a learned map is converting a hidden state $(g_i)_t$ to a new hidden state using one more piece of source data. The hidden state can store a representation which knows about everything which has already been seen.

If we need to output one value for the whole sequence, the highest levels of the RNN will be fully connected, initialised from just the final timestep of the last recurrent layer, i.e. $(g_i)_{u_i^1}$.

5.4.4 LSTM

When trying to use RNNs when the sequence is long, it is hard for the early parts (low t) of the sequence to influence the final state, and thus there is a limit to what the network can learn. The state has to be remembered through many multiplications by the matrix $\bar{W}_i$. For most values of $\bar{W}_i$ the derivative of $(g_i)_{u_i^1}$ with respect to an element in $(g_{i-1})_t$ for $t \ll u_u^1$ will be very large or very small. This is called the problem of exploding or vanishing gradients.

Long-short term memory (LSTM, [HS97]) is a variation of a recurrent layer which is designed to allow information to survive in the state across many timesteps. There are two separate sets of units which operate in pairs. To each output element $\big((g_i(x,\lambda))_t\big)_j$ is associated a memory cell $(c_t)_j := \big((c_i(x,\lambda))_t\big)_j$. The layer needs parameters[2] $u_{i-1} \times u_i$ matrices W_i^f, W_i^i, W_i^o and W_i^c, $u_i \times u_i$ matrices $\bar{W}_i^f$, $\bar{W}_i^i$, $\bar{W}_i^o$ and $\bar{W}_i^c$, and u_i-vectors b_i^f, b_i^i, b_i^o and b_i^c. The layer uses the sigmoid function, $\sigma'(x) = \frac{1}{1+e^{-x}}$ which is monotone increasing from $\mathbb{R}$ to the interval $(0,1)$. The normal activation function, σ, is usually tanh. They are evaluated for $t > 1$ using temporary variables f_t, i_t, o_t and $\tilde{c}_t$ as follows.

$$\begin{aligned}
f_t &= \sigma'((g_i(x,\lambda))_t W_i^f + (g_{i+1}(x,\lambda))_{t-1}\bar{W}_i^f + b_i^f) && (5.8)\\
i_t &= \sigma'((g_i(x,\lambda))_t W_i^i + (g_{i+1}(x,\lambda))_{t-1}\bar{W}_i^i + b_i^i) && (5.9)\\
o_t &= \sigma'((g_i(x,\lambda))_t W_i^o + (g_{i+1}(x,\lambda))_{t-1}\bar{W}_i^o + b_i^o) && (5.10)\\
\tilde{c}_t &= \sigma((g_i(x,\lambda))_t W_i^c + (g_{i+1}(x,\lambda))_{t-1}\bar{W}_i^c + b_i^c) && (5.11)\\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && (5.12)\\
(g_{i+1})_t &= o_t \odot \sigma(c_t) && (5.13)
\end{aligned}$$

where $\odot$ denotes the Hadamard/entrywise product. To interpret these formulae for $t = 1$, we think of c_0 and $(g_{i+1})_0$ as being 0.

It is hard to know exactly why a neural network is working, but there are aspects of the design of LSTM which make sense. The usual interpretation of this is as follows. We think of values which have passed through σ' as being practically boolean 0/1 values. The c_t cells preserve information for many time steps. A cell can be turned off, or made to ignore its existing value, by some timestep triggering the "forget gate", which means setting f_t to 0[3], and can be given a new value, the calculated $\tilde{c}_t$ by the "input gate" i_t being triggered. Whether the value in a cell is actually used for output at a timestep is governed by the "output gate" o_t. We can see that it is easy for a value in x_t to affect a cell in $c_{t'}$ with $t \ll t'$ because it could be stored in some cell c_t and then not be forgotten. This is how the vanishing gradient problem is alleviated.

[2]Note that in this widely accepted notation, the **superscripts** i, o and f are decorations not variables nor placeholders.

[3]A model where f is "whether to forget", with f_t replaced by $(1 - f_t)$ in (5.12), would be totally equivalent.

The long term information stored in the cells is separate from the immediate information stored in the outputs. Note that if f manages to stay 0 and o and i manage to stay 1 then this is a vanilla RNN unit, albeit with the nonlinearity $\tanh \circ \tanh$, see Figure 5.2.

The only link between $(c.)_j$ and $(g.)_j$ (which means that the cell and the hidden state of a unit are linked, and we don't just have a load of hidden states and a load of cells) is the final equation.

It is not clear that tanh is needed in both places, and it could easily be replaced with another activation function.

LSTM achieves impressive results in practice, though can take a long time to train. There are many variations on LSTM which have been proposed. It requires a very large amount of experimentation to be sure which modifications are real improvements. There is also recent suggestion (e.g. [BKK18] and [MH18]) that for many current applications recurrent networks are not needed at all, because for many tasks their performance can be beaten by modern designs of 1D-CNNs, which are more efficient to train.

5.4.5 Dropout and batch normalization

There are two very useful methods for improving the effectiveness of neural networks. They have in common that although they do not change (or hardly change) the form of $\tilde{f}$, a variant of $\tilde{f}$ is actually used during training.

In dropout [Hin+12; Sri+14], we may pick a *dropout probability* $p \in (0, 1)$, typically 0.5, for a weight matrix W somewhere in the definition of $\tilde{f}$. Then, for each training example $x \in X$ used in training we generate a matrix B of i.i.d. Bernoulli(p) random variables and replace $\tilde{f}(x, \lambda)$ with its value where W has been replaced with $\frac{1}{p} W \odot B$, where $\odot$ denotes the elementwise or Hadamard product. (In [GRR15] we suggest picking a single B for a whole minibatch, which is an additional optimization to consider.) The effect of this tends to be that learning is slower but generalisation is better. This is thought to be partly because two units cannot train together.

Applying batch normalisation [IS15] to a scalar value z somewhere in $\tilde{f}$ means adding two scalars β and γ to the set of parameters to learn, Λ, and replacing z in the definition of $\tilde{f}$ with $\gamma \frac{z - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}} + \beta$ where ε is a small fixed parameter. During training, $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ are the mean and standard deviation of z across the minibatch being used (which will depend on the current values of both χ and λ), and during testing/inference they are estimates of z across the whole training set. Normally this will be applied across all the activations in a layer, possibly for every layer. In the case of convolutional and recurrent layers, the parameters β and γ for all the units which share the same weight will usually be shared. For example the same unit being calculated at different places in the input will share these values. That way the symmetrical structure of the layer is not broken, and the same unit always sees the same type of input.

In the experiments our focus is on the use of the signature, and I've kept the network architectures very simple. In practical applications, these two techniques are likely to be very important for getting the best results.

5.4.6 Initialisation

The algebraic formulae defining a neural network model are unchanged if they are composed with an affine transform of the basic data: this is just equivalent to changing the weights and biases. In training the weights are initialised only one way, however. As a consequence, if extreme translations of the data are necessary to find meaningful patterns, these translations will take a long time to be found. In order for all the units to have a chance to be useful, it makes sense to ensure that all the inputs are about the same size (e.g. around $[-1, 1]$ or $[0, 1]$) and to initialise the random weight matrices to keep this approximate scale for all units. Much effort has been put into making this work well, famously [GB10; He+15; MM15].

When classifying Chinese characters, which are all on the same scale, I might want to use coordinates as features. We rescale the bounding box to $[-1, 1]^2$ so that these coordinates can all be in a sensible range.

Elements of the signature or log signature of a path at level m scale like length^m when the path is scaled. When using the signature as input to a neural network the scaling has to be taken into account. Once a feature is chosen, an easy approach is to find the mean and variance of that feature over a large random sample of training data and then use the mean and variance to always scale the feature to aim at a mean of 0 and a variance of 1. Others have had other approaches to this. One suggested by [LJY17], around equation (5), is equivalent to enlarging the path to a fixed length before taking the signature.

5.5 Symmetries for handwriting

In online character recognition each character is given as a sequence of strokes, each stroke is given as a sequence of coordinate points. In our work, we regard these points just as a 2D path. That means we are deliberately ignoring other facts about the data which may come from the pen, for example the time of each coordinate point and the force in the pen. For readers, and therefore potentially in the education of writers, the time spent on each part of the stroke is not important, so ignoring this is a good idea. There are usually many more given points in a stroke in each important turn. Given this as our approach, there is an approximate invariant, in that single points can be added or removed along the path and the character will remain similar. If the stroke's signature is taken as the only representation of the stroke, then we have a method which automatically works with this invariant. This is a consequence of the fact that the signature of a path is independent of its parametrisation. The other methods of representation I have attempted needed to explicitly bear in mind this invariant.

Chinese characters stand alone enough that it is meaningful to want to classify them individually. The absolute size of a character is generally not important for distinguishing Chinese characters. (There could be scripts where this is not the case.) We therefore scale the character, preserving aspect ratio, to occupy a standard square, in particular $[-1,1]^2$. This means that we will have no trouble with writers who chose to write in different sizes.

Small distortions in how a path looks are invariants for characters. Data augmentation is needed to take advantage of this, and doing so is needed to get good results on the CASIA data, but experimenting with augmentation is not my aim. I stick with a single method of augmentation which performs the following to each character, which works well, and which I inherited from Ben Graham. The character is enlarged by a separate uniformly random factor in $[0.7, 1.3]$ in the vertical and horizontal directions. Then α is picked uniformly random in $[-0.3, 0.3]$ and one of the following three things is picked to happen uniformly at random: (1) the character is rotated by α radians, (2) the character is horizontally sheered by α, (3) the character is vertically sheered by α. Then the character is scaled to lie within a fixed bounding box.

5.6 Chinese handwriting recognition results

Here I present some highlights of experiments attempting to learn the CASIA data with signatures and RNNs. The aim is to do well at this complex path classification task using a fast, smaller model. A character is not a single path, it is a series of paths, one for each stroke. We come up with some way to use the signature to form a representation of a character which we feed to a traditional classifier. Training as we did on a single desktop with a single graphics processing unit (GPU,)[4] it

[4]Originally designed for graphics, a piece of commodity hardware which is commonly and effectively used for training neural networks in addition to the central processing unit(s) (CPU) in each computer.

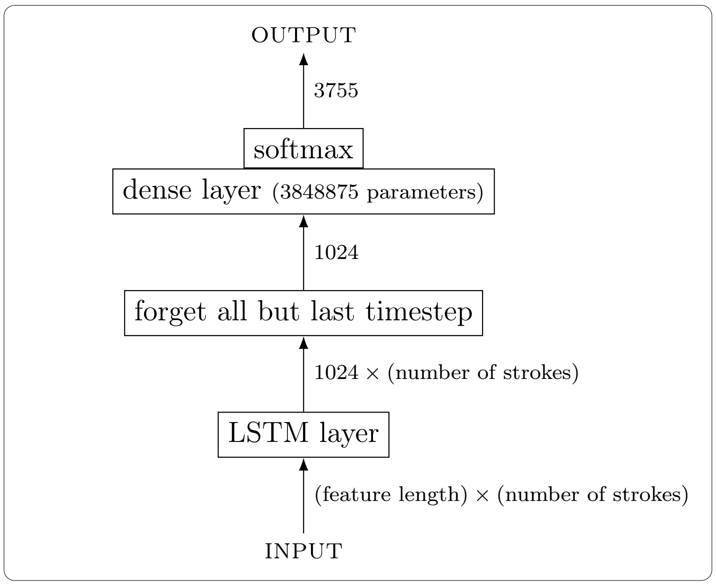

OUTPUT
3755
softmax
dense layer (3848875 parameters)
1024
forget all but last timestep
1024 × (number of strokes)
LSTM layer
(feature length) × (number of strokes)
INPUT

Figure 5.3: Schematic diagram of LSTM network architecture for signatures of each stroke. The data shapes are indicated on the arrows.

is generally the case that the training time is dominated by the neural network operations on the GPU. Calculating minibatches with these representations on CPU-only background threads, for example using `iisignature`, was fast enough that the GPU was continuously using the minibatches for training. This indicates that further optimising the signature calculations would not significantly speed up this training.

5.6.1 Signatures of each stroke

The simplest method is to make a representation of each stroke and use the sequence of these representations as the input to the RNN. The signature alone would not make a good representation, as the network would not then have any indication of the relative locations of the strokes, which is important. We therefore use the starting point and the signature as the representation. We feed this to a simple 1-layer LSTM, with the architecture shown in Figure 5.3. We trained the network with the common method Adam [KB15] with its default parameters.

Because all the strokes are not much longer than a line in the bounding box, it was not necessary to normalise the signatures. In all these cases, the network was fully trained after 5 epochs which took about an hour. The final test accuracies are shown in Table 5.1. We see that the signature contains some useful information about the stroke. The method does not reach anywhere near useful accuracy.

Signature level	representation length	Total parameters	test accuracy
none	2	8 055 467	0.267
1	4	8 063 659	0.493
2	8	8 080 043	0.576
3	16	8 112 811	0.630
4	32	8 178 347	0.658

Table 5.1: Results summary on CASIA1.1 of training LSTM on signatures of each stroke

5.6.2 Signatures of local segments

This is a simple and more successful representation method of a character, which first converts the character to a single 3-dimensional path. I discussed a few methods for making a single 3-dimensional path from a character in [Rei14]. A basic method is to add a "pen dimension" which is 0 on each stroke (pen down) and 1 on an added straight line (pen up) between the endpoints of the strokes. These 3D paths vary in their total length. A basic step is to chop the paths into segments of equal length. This length includes the length expended in the pen direction, otherwise the interpolation can produce weird artefacts. This makes it a bit more likely that a split in a path will happen at a pen lifting. We experimented with overlapping the segments, but this was not clearly advantageous. We use this split of the character as the time dimension of the RNN, picking some representation of each segment. Different characters will have different lengths, but the LSTM requires a whole minibatch to have the same time length, so we pad backwards in time with zeros.

A simple and effective representation is the starting point of the segment concatenated with the signature of the 3-dimensional path which is that segment up to some level. We considered adding the bounding box or centroid of that segment but it did not obviously help.

In these comparisons, we split the characters into segments (after scaling) of length 0.4. Different characters have different numbers of segments, with the maximum being around 115. The architecture is just the same as for the signatures of strokes case, except the time dimension is now "number of segments" note "number of strokes". Again, we used Adam with default parameters and we were able to do these experiments without scaling the inputs. Beyond level 4 the representation of a character is far too large to learn from efficiently. Level 4 is already rather large. We observe that this method is promising.

Signature level	representation length	Total parameters	time (hr)	test accuracy
none	3	8 059 563	14.9	0.909
1	6	8 071 851	15.6	0.933
2	15	8 108 715	15.6	0.945
3	42	8 219 307	15.8	0.951
4	123	8 551 083	18.5	0.947

Table 5.2: Results summary on CASIA1.1 of training LSTM on signatures of local segments. Training was for 10 epochs.

5.7 Sketchnet

The Sketchnet dataset [EHA12] consists of hand-sketches of each of 250 classes of objects, like 'tomato', 'banana' and 'TV' by 80 different writers. There is no standard train/test split, I take the first 40 of each for training and the last 40 for testing. This isn't comparable with others. Each stroke in the Sketchnet data is not given just as a series of points, but often includes a series of Bezier curves. The data thus appears to have been coarsened. I interpolate each given curve with ample points before doing anything else with the data, so that it is like the situation with CASIA data.

Unlike characters of handwriting, a flip in a vertical axis is a reasonable invariant for sketches and so I include such a flip with probability $\frac{1}{2}$ in the data augmentation.

There has been much work on this type of human sketch recognition, most prominently Google's "Quick, draw"[5] which went viral in 2017. The methods used were not released, but when I saw this work I knew they had much more data and much more accuracy than I could hope for. Using a similar signatures of local segments method as for the RNNs with CASIA, and with very little tweaking, I got to a test error of 33%.

5.8 Signatures in LSTM

Making models which are more efficient than LSTMs is something many people are trying. LSTMs can take a long time to train, and although they perform very impressively they seem logically inefficient for a couple of reasons to my mind.[6] First, that they have to learn these operations of "whether to save" and "whether to forget"

[5] https://quickdraw.withgoogle.com

[6] The thinking here is my woolly intuition, and is suffused with the traditional, perhaps lazy, anthropomorphism.

individually, which is quite remote from the actual problem being solved. Second, that if the thing they need to save is high dimensional they have to teach themselves into a state which allows a group of cells to cooperate. We know that signatures provide a good summary of the shape of a path, and the history of something can sometimes be thought of as a path, and networks memory is just its choice of salient features of its memory, so we wondered whether we could use signatures as this memory. The signature would be an input to a cell of a RNN based on the history of that cell. Note in particular that *this* effort is not trying to produce a network which is specially designed to handle curves or which is taking signatures of data as input, but rather we are trying to come up with something which solves the sort of problems a vanilla LSTM is used for, and is potentially a drop-in replacement, but is ultimately, hopefully, better in some way – faster to train or smaller or more robust. The signatures are taken internally in the network; the user would not need to know about them. In this effort the derivatives of the signature are necessary because there are parameters in the calculation graph of the network which are affecting the input of signature calculations. We tried several architectures to try to test out this type of idea, for example where a cell was given a signature of the graph of its activation through history as an input.

Only one type of architecture was successful in the most minimal way, namely that we could train the network to solve a toy problem. This was suggested by Harald Oberhauser, and involves following the structure of the LSTM as closely as possible. For example, we want to separate the internal memory from the output of a layer, and we want to separate the space where the memory lives from the spaces of our layer's inputs and outputs. The idea is that, for some K and m, the memory is a signature of a K-dimensional path up to level m, i.e. a value in a truncated tensor space $T^{\underline{m}}(\mathbb{R}^K)$. We are not defining it by specifying the path of which it is the signature. We choose a configuration which will be the same as vanilla LSTM when $m = 1$ but generalises it for $m > 1$.

The way we forget is to collapse the signature in a chosen direction, either entirely or partially. The matrix of stretching by a factor λ in the direction of the unit vector $\hat{\imath}$ is $(I_K - (1-\lambda)\hat{\imath}\hat{\imath}^T)$. If we were to generate $\hat{\imath}$ on its own in the network by making a vector $\vec{\imath}$ and normalising it there would be a nasty discontinuity around small $\vec{\imath}$. It would be nicer to do something like using $\|\vec{\imath}\|$ to get $1-\lambda$ or (which looks nicer) $\sqrt{1-\lambda}$ so that small values of $\vec{\imath}$ don't do anything. But we don't want $1-\lambda$ to be bigger than 1. So we use $\tanh\|\vec{\imath}\|$ as $\sqrt{1-\lambda}$ to get the matrix $\left(I_K - \frac{\tanh^2\|\vec{\imath}\|}{\|\vec{\imath}\|^2}\vec{\imath}\vec{\imath}^T\right)$. To apply this transformation to the signature we premultiply each level m' by the matrix's m'th Kronecker-product power. Call $\vec{\imath}$ the *forget-vector*, and this operation on signatures $\text{Forget}_{\vec{i}}(\cdot)$.[7]

To save information, we concatenate a new segment onto the path – i.e. Chen product the stored signature with the signature of a chosen segment.[8] We need to pick the new segment in a sensible way to maintain the scaling of the signature, so we always make the new segment be a unit vector. We let $\text{Add}_{\vec{f}}(S)$ be the signature obtained as the Chen product of S and $\exp(\frac{1}{\|\vec{f}\|}\vec{f})$

We call one of these stored signatures a *signature cell*. There are clearly many variations which could be tried, but for these experiments we have only one of them in the whole layer. We call the number of hidden units in the layer u^0_{i+1}.

We can't have an equation like (5.12) for a signature cell because a linear combination of signatures doesn't produce a signature. A simple way to do some forgetting and adding is to always do the forgetting and then always concatenate something on.

How to generate output based on the signature – in analogy with (5.13) – is not obvious. Having a parameter the shape of a signature and using it as a linear functional is out solution to this, although it does involve adding many extra parameters.

Putting things together, we have the following parameters for an input to our layer i of shape $u^0_i \times u^1_i$. $K \times u^1_i$ weights matrices W^f and W^i, $K \times u^0_{i+1}$ weights matrices $\bar{W}^f$ and $\bar{W}^i$, and length-K biases b^f and b^i. $u^0_{i+1} \times u^1_i$ weights matrix W^o, $u^0_{i+1} \times u^0_{i+1}$ weights matrix $\bar{W}^o$, and length-u^0_{i+1} biases b^o and b^h (if S doesn't include a fixed 1). Also u^0_{i+1} signature-shaped parameters V^h each of size $\frac{u^0_{i+1}((u^0_{i+1})^m-1)}{u^0_{i+1}-1}$ – another big matrix. We used a common scheme for initialising the parameters – the biases were initialised at zero, and each weight matrix was initialised in the "Glorot Uniform" method built in to Keras – each element being uniformly distributed in $[-\frac{\sqrt{6}}{g}, \frac{\sqrt{6}}{g}]$ where g is the sum of the matrix's last two dimensions. The following are the complete defining equations, note that the variables p and p' vary among the

[7]In practice, this uses the `sigscale` functionality of `iisignature`.

[8]In practice, this uses the `sigjoin` functionality of `iisignature`.

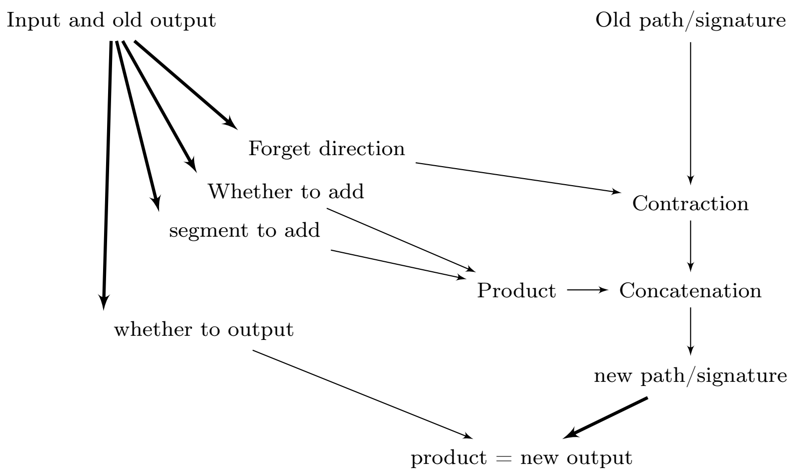


Figure 5.4: Schematic of one cell of an LSTM layer with signature memory. Bold arrows indicate that learned parameters are involved.

u^0_{i+1} output units.

$$i_{kt} = \sum_l W^i_{kl}((g_i)_t)_l + \sum_{p'} \bar{W}^i_{kp'} h_{p',t-1} + b^i_k \qquad k\text{th element of forget vector } \vec{i}_t \tag{5.14}$$

$$f_{kt} = \sum_l W^f_{kl}((g_i)_t)_l + \sum_{p'} \bar{W}^f_{kp'} h_{p',t-1} + b^f_k \qquad k\text{th element of new information vector } \vec{f}_t \tag{5.15}$$

$$o_{pt} = \sigma\left(\sum_l W^o_{pl}((g_i)_t)_l + \sum_{p'} \bar{W}^o_{pp'} h_{p',t-1} + b^o_p\right) \qquad \text{in [0,1], whether to output anything} \tag{5.16}$$

$$S_t = \mathrm{Add}_{\vec{f}_t}(\mathrm{Forget}_{\vec{i}_t}(S_{t-1})) \tag{5.17}$$

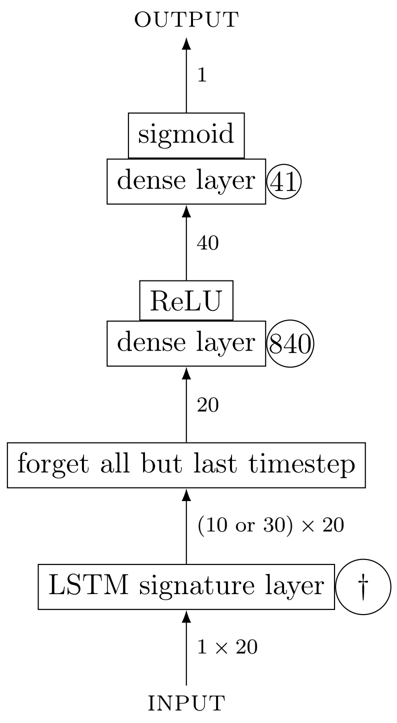
$$\tilde{h}_{pt} = V^h_p \cdot S_t + b^h_p \qquad \text{dot product of signatures as potential output} \tag{5.18}$$

$$h_{pt} = o_{jt} \tanh(\tilde{h}_{pt}) \qquad \text{output} \tag{5.19}$$

The structure of a single timestep of one of these LSTM signature layers is summarised in Figure 5.4.

5.8.1 Toy problem

We wanted a very simple example problem to see if our new design of network could (a) learn anything and (b) have memory, i.e. learn something which depends only on a long-term dependency in the data. We wanted the problem to be simple as the calculation is quite slow despite using `iisignature`, because significantly it is



$K = u_1^0$	m	†
10	1	580
10	2	1580
10	3	11580
30	1	4740
30	2	31740
30	3	841740

Figure 5.5: Schematic diagram of architecture used for training an LSTM signature layer on a toy model. The data shapes are indicated on the arrows. Numbers of parameters are indicated in circles.

happening on a CPU rather than a GPU. We came up with the following task, which can also easily be solved with a traditional LSTM. The task is to distinguish between (category A) a random binary string of i.i.d. Bernouilli(0.5) zeros and ones, and (category B) a string which begins with a shorter length v binary string and continues with the length v partial sums mod 2. For example, the following are such strings with $v = 4$.

$$0010\ 1001001001$$

$$1011\ 1101111011$$

Distinguishing these requires remembering something for at least v steps. Also it is not possible to distinguish the classes perfectly, because a random string may have the form of the other strings by chance, but this is a small effect. For the experiments I had a single one of my LSTM-signature layers, of which the value of the timestep was passed to a dense layer with 40 ReLU units, which was connected to a single sigmoid output. A summary of the network in use is shown in Figure 5.5.

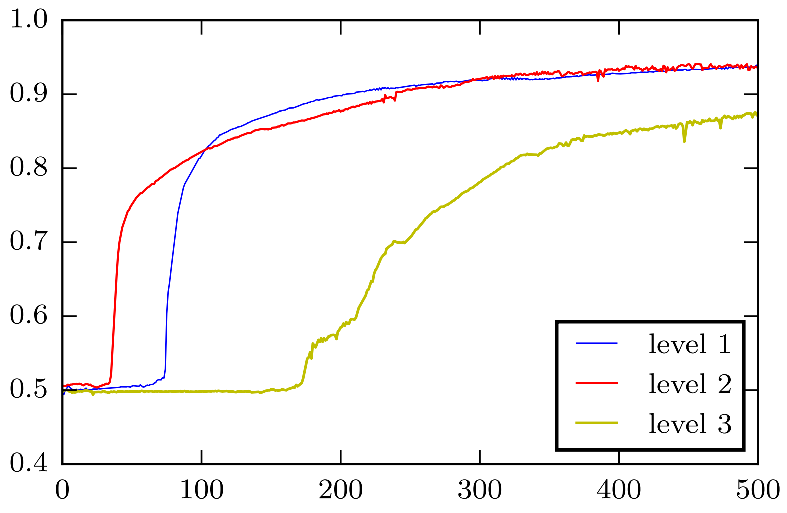


Figure 5.6: Accuracy of the classification of training the LSTM Signature network on the toy problem after each epoch. The first 15 of the 20 characters are random. Signature level: blue 1, red 2, yellow 3. 10 dimensions with 10 hidden units.

Because this was just a proof-of-concept and unlimited data could be generated, there was no need to have separate train and test sets. We just had a very large training set, which is highly representative of the two classes. The general result was that these networks were able to learn, but took a long time to start learning in some cases. This seems to be because initialisation is hard to get right. The graphs show the accuracy over the whole training set of 11000 samples after each epoch (pass through the whole of the data). In the experiments which I plotted, v is 15 and so 15 of the $u_0 = 20$ units are always random, so it is not easy to memorise the members of category B and memory of length 15 steps is requires. In Figure 5.6 I show cases for $m = 1, 2, 3$ for $K = u_1^0 = 10$, and in Figure 5.7 I show cases for $K = u_1^0 = 30$.

There is a lot of scope for playing with this model — many parameters to tweak and various initialisations to try. If investing heavily into this, I would want to move the calculations onto a GPU for much more efficiency, and then try to learn a standard sequence modelling benchmark, such as the famous Penn Treebank. The success of this experiment is in showing that something can be learnt by a layer with a signature inside.

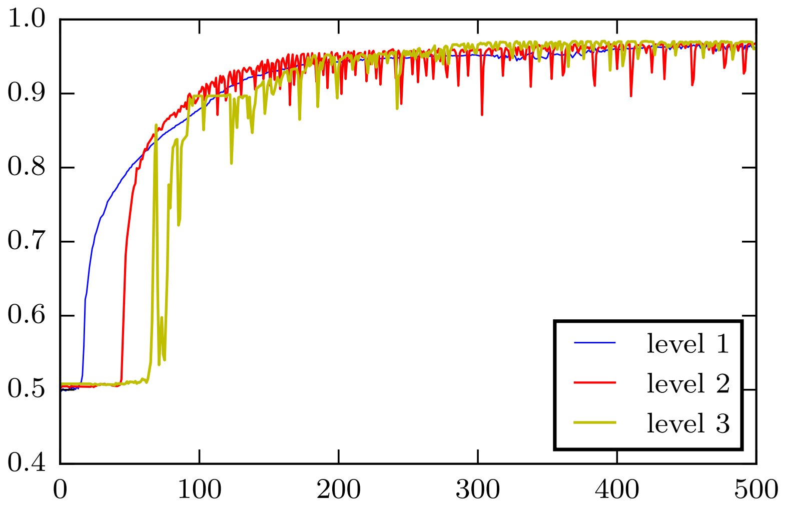


Figure 5.7: Accuracy of the classification of training the LSTM Signature network on the toy problem after each epoch. 15 of the 20 characters are random. Signature level: blue 1, red 2, yellow 3. 30 dimensions with 30 hidden units.

Symbols and abbreviations index

Bibliography

[AA98] Ethem Alpaydin and Fevzi Alimoglu. *Pen-Based Recognition of Handwritten Digits Data Set*. 1998. URL: https://archive.ics.uci.edu/ml/datasets/Pen-Based+Recognition+of+Handwritten+Digits (cit. on pp. 4, 5).

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning ways for your mathematical plays (vol 1)*. 1982 (cit. on p. 39).

[BD93] Roger W. Brockett and Liyi Dai. “Non-holonomic Kinematics and the Role of Elliptic Functions in Constructive Controllability”. In: *Nonholonomic Motion Planning*. Ed. by Zexiang Li and J. F. Canny. Boston, MA: Springer US, 1993, pp. 1–21 (cit. on p. 18).

[BH05] Zhen-Long Bai and Qiang Huo. “A study on the use of 8-directional features for online handwritten Chinese character recognition”. In: *Eighth International Conference on Document Analysis and Recognition, 2005. Proceedings*. Aug. 2005, 262–266 Vol. 1 (cit. on p. 76).

[BKK18] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling”. In: (2018). URL: http://arxiv.org/abs/1803.01271 (cit. on p. 87).

[Blü04] Johannes Blümlein. “Algebraic relations between harmonic sums and associated quantities”. In: *Computer Physics Communications* 159.1 (May 2004), pp. 19–54. URL: https://arxiv.org/abs/hep-ph/0311046 (cit. on p. 54).

[Che58] Kuo-Tsai Chen. “Integration of Paths – A Faithful Representation of Paths by Noncommutative Formal Power Series”. In: *Transactions of the American Mathematical Society* 89.2 (1958), pp. 395–407. URL: https://www.jstor.org/stable/1993193 (cit. on pp. 8, 49).

[CK16] Ilya Chevyrev and Andrey Kormilitzin. “A Primer on the Signature Method in Machine Learning”. 2016. URL: http://arxiv.org/abs/1603.03788 (cit. on pp. 2, 5).

[CM] Fernando Casas and Ander Murua. *The BCH formula and the symmetric BCH formula up to terms of degree 20.* URL: http://www.ehu.eus/ccwmuura/bch.html (cit. on p. 51).

[CM09] Fernando Casas and Ander Murua. “An efficient algorithm for computing the Baker-Campbell-Hausdorff series and some of its applications”. In: *Journal of Mathematical Physics* 50.3 (Mar. 2009), p. 033513. eprint: http://arxiv.org/abs/0810.2656 (cit. on pp. 51, 53).

[CMS12] Dan Cireşan, Ueli Meier, and Jurgen Schmidhuber. “Multi-column Deep Neural Networks for Image Classification”. In: *Proc. 2012 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. CVPR ’12. IEEE Computer Society, 2012, pp. 3642–3649 (cit. on p. 76).

[Cub+18] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. “AutoAugment: Learning Augmentation Policies from Data”. In: (2018). URL: https://arxiv.org/abs/1805.09501 (cit. on p. 82).

[DFP92] Persi Diaconis, James Allen Fill, and Jim Pitman. “Analysis of Top To Random Shuffles”. In: *Combinatorics, Probability and Computing* 1.2 (1992), pp. 135–155 (cit. on p. 46).

[Die13] Joscha Diehl. “Rotation Invariants of Two Dimensional Curves Based on Iterated Integrals”. 2013. URL: http://arxiv.org/abs/1305.6883 (cit. on pp. 10, 37, 38, 40).

[DIM18] Askar Dzhumadil’daev, Nurlan Ismailov, and Farukh Mashurov. “Embeddable algebras into Zinbiel algebras via the commutator”. 2018. URL: https://arxiv.org/abs/1809.10550 (cit. on p. 44).

[DR18] Joscha Diehl and Jeremy Reizenstein. “Invariants of multidimensional time series based on their iterated-integral signature”. In: *J. Acta Appl Math* (2018). URL: https://rd.springer.com/article/10.1007%2Fs10440-018-00227-z (cit. on pp. vii, 24).

[EHA12] Mathias Eitz, James Hays, and Marc Alexa. “How Do Humans Sketch Objects?” In: *ACM Trans. Graph. (Proc. SIGGRAPH)* 31.4 (2012), 44:1–44:10. URL: http://cybertron.cg.tu-berlin.de/eitz/projects/classifysketch/ (cit. on p. 92).

[EN11] Alexander Engström and Patrik Norén. “Polytopes from subgraph statistics”. In: *Discrete Mathematics & Theoretical Computer Science* (2011). URL: http://arxiv.org/abs/1011.3552 (cit. on p. 37).

[FKK10] Shuo Feng, Irina Kogan, and Hamid Krim. “Classification of Curves in 2D and 3D via Affine Integral Signatures”. In: *Acta Applicandae Mathematicae* 109.3 (Mar. 2010), pp. 903–937 (cit. on pp. 22, 23, 25).

[FV10] Peter K. Friz and Nicolas B. Victoir. *Multidimensional Stochastic Processes as Rough Paths: Theory and Applications*. 2010 (cit. on pp. 18, 33, 34, 37).

[FW86] Joy P. Fillmore and S. Gill Williamson. “Permanents and determinants with generic noncommuting entries”. In: *Linear and Multilinear Algebra* 19.4 (1986), pp. 321–334 (cit. on p. 25).

[Gal63] David Gale. “Neighborly and cyclic polytopes”. In: *Proc. Sympos. Pure Math.* 7 (1963) (cit. on p. 36).

[Gar12] Adriano Garsia. *On the powers of top to random shuffling*. 2012 (cit. on p. 46).

[GB10] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html (cit. on p. 88).

[Gil17] Mike Giles. *Private communication via Terry Lyons*. 2017 (cit. on p. 53).

[GK08] Eric Gehrig and Matthias Kawski. “A Hopf-Algebraic Formula for Compositions of Noncommuting Flows”. In: *Proceedings of the IEEE Conference on Decision and Control*. 2008, pp. 1569–1574 (cit. on p. 12).

[GK14] Jean-Paul Gauthier and Matthias Kawski. “Minimal complexity sinusoidal controls for path planning”. In: *53rd IEEE Conference on Decision and Control*. Dec. 2014, pp. 3731–3736 (cit. on p. 18).

[GMW10] Oleg Golubitsky, Vadim Mazalov, and Stephen M. Watt. “Toward Affine Recognition of Handwritten Mathematical Characters”. In: *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*. DAS ’10. Boston, Massachusetts, USA: ACM, 2010, pp. 35–42. ISBN: 978-1-60558-773-8 (cit. on p. 22).

[GOT17] Jacob E. Goodman, Joseph O'Rourke, and Csaba D. Tóth. *Handbook of Discrete and Computational Geometry*. 2017 (cit. on p. 35).

[Gra13] Benjamin Graham. "Sparse arrays of signatures for online character recognition". 2013. URL: `http://arxiv.org/abs/1308.0371` (cit. on pp. 2, 76).

[Gri18] Darij Grinberg. *Is this sum of cycles invertible in QSn? - MathOverflow*. 2018. URL: `https://mathoverflow.net/questions/308536` (cit. on p. 46).

[GRR15] Benjamin Graham, Jeremy Reizenstein, and Leigh Robinson. "Efficient batchwise dropout training using submatrices". 2015. URL: `http://arxiv.org/abs/1502.02478` (cit. on p. 87).

[Hal50] Marshall Hall. "A basis for free Lie rings and higher commutators in free groups". In: (1950). URL: `http://www.ams.org/journals/proc/1950-001-05/S0002-9939-1950-0038336-7/home.html` (cit. on p. 10).

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: (2015). URL: `http://arxiv.org/abs/1502.01852` (cit. on p. 88).

[Hin+12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors". 2012. URL: `http://arxiv.org/abs/1207.0580` (cit. on p. 87).

[HL10] Ben Hambly and Terry Lyons. "Uniqueness for the signature of a path of bounded variation and the reduced path group". In: *Annals of Mathematics* 171.1 (2010), pp. 109–167 (cit. on p. 3).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. URL: `http://dx.doi.org/10.1162/neco.1997.9.8.1735` (cit. on p. 86).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 448–456. URL: `http://proceedings.mlr.press/v37/ioffe15.html` (cit. on p. 87).

[Jin] Jin, Lianwen et al, Human-Computer Intelligent Interaction Lab, South China University of Technology. *SCUT GPEN*. URL: hcii-lab.net/gpen/ (cit. on p. 77).

[KB15] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* abs/1412.6980 (2015). URL: http://arxiv.org/abs/1412.6980 (cit. on p. 90).

[KS53] Samuel Karlin and Lloyd S Shapley. *Geometry of moment spaces*. 1953 (cit. on p. 35).

[LCL07] Terry Lyons, Michael Caruana, and Thierry Lévy. *Differential Equations Driven by Rough Paths*. 2007 (cit. on pp. 8, 9).

[Lee91] C. Lee. "Regular triangulations of convex polytopes". In: *Applied Geometry and Discrete Mathematics: The Victor Klee Festschrift*. Ed. by Bernd Sturmfels and Peter Gritzmann. 1991 (cit. on p. 35).

[Liu+10] Cheng-Lin Liu, Fei Yin, Da-Han Wang, and Qiu-Feng Wang. "Chinese Handwriting Recognition Contest 2010 (report)". 2010. URL: www.nlpr.ia.ac.cn/2010papers/gnhy/nh5.pdf (cit. on p. 76).

[Liu+11] Cheng-Lin Liu, Fei Yin, Da-Han Wang, and Qiu-Feng Wang. *CASIA Online and Offline Chinese Handwriting Databases*. 2011. URL: http://www.nlpr.ia.ac.cn/databases/handwriting/Home.html (cit. on p. 77).

[LJY17] Songxuan Lai, Lianwen Jin, and Weixin Yang. "Online Signature Verification using Recurrent Neural Network and Length-normalized Path Signature". In: (2017). URL: http://arxiv.org/abs/1705.06849 (cit. on p. 88).

[Lod01] Jean-Louis Loday. "Dialgebras". In: *Dialgebras and Related Operads*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 7–66 (cit. on p. 42).

[Lod95] Jean-Louis Loday. "Cup-Product for Leibniz Cohomology and Dual Leibniz Algebras". In: *Mathematica Scandinavica* 77.2 (1995), pp. 189–196 (cit. on p. 42).

[LR95] Pierre Lalonde and Arun Ram. "Standard Lyndon Bases of Lie Algebras and Enveloping Algebras". In: *Transactions of the American Mathematical Society* 347.5 (1995), pp. 1821–1830 (cit. on p. 10).

[LS06] Terry J. Lyons and Nadia Sidorova. “On the Radius of Convergence of the Logarithmic Signature”. In: *Illinois J. Math.* 50.1-4 (2006), pp. 763–790 (cit. on p. 9).

[Lyo+10] Terry Lyons, Stephen Buckley, Djalil Chafai, Greg Gyurkó, and Arend Janssen. *CoRoPa Computational Rough Paths (software library)*. 2010. URL: `http://coropa.sourceforge.net/` (cit. on pp. 11, 49, 57, 59).

[Lyo98] Terry Lyons. “Differential equations driven by rough signals”. In: *Revista Matemática Iberoamericana* 14 (1998), pp. 215–310 (cit. on p. 2).

[Meu+17] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103 (cit. on p. 17).

[MH18] John Miller and Moritz Hardt. “When Recurrent Models Don’t Need To Be Recurrent”. In: (2018). URL: `http://arxiv.org/abs/1805.10369` (cit. on p. 87).

[MM15] Dmytro Mishkin and Jiri Matas. “All you need is a good init”. In: (2015). URL: `http://arxiv.org/abs/1511.06422` (cit. on p. 88).

[NWF06] Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. “Building Workload Characterization Tools with Valgrind”. In: *IEEE International Symposium on Workload Characterization* (2006). URL: `http://valgrind.org/docs/iiswc2006.pdf` (cit. on p. 65).

[PSS18] Max Pfeffer, Anna Seigal, and Bernd Sturmfels. “Learning Paths from Signature Tensors”. 2018. URL: `http://arxiv.org/abs/1809.01588` (cit. on p. 18).

[Ree58] Rimhak Ree. “Lie Elements and an Algebra Associated With Shuffles”. In: *Annals of Mathematics* 68.2 (1958), pp. 210–220 (cit. on p. 9).

[Rei14] Jeremy Reizenstein. “Signatures in online handwriting recognition. MSc miniproject”. 2014. URL: `https://github.com/bottler/phd-docs/blob/master/handwriting.pdf` (cit. on p. 91).

[Rei15] Jeremy Reizenstein. “Calculation of Iterated-Integral Signatures and Log Signatures”. 2015. URL: `http://arxiv.org/abs/1712.02757` (cit. on pp. 7, 14, 58).

[Reu94] Christophe Reutenauer. *Free Lie Algebras*. 1994 (cit. on pp. 7, 9–12, 16, 53, 55, 71).

[RG18] Jeremy Reizenstein and Benjamin Graham. “The iisignature library: efficient calculation of iterated-integral signatures and log signatures”. 2018. URL: `http://arxiv.org/abs/1802.08252` (cit. on p. vii).

[Sch58] Marcel P. Schutzenberger. "Sur une propriété combinatoire des algèbres de Lie libres pouvant être utilisée dans un problème de mathématiques appliquées". In: *Séminaire Dubreil, Algèbre et théorie des nombres* 12.1 (1958), pp. 1–23. URL: http://www.numdam.org/article/SD_1958-1959__12_1_A1_0.pdf (cit. on p. 43).

[Shi53] Anatoli Illarionovich Shirshov. "Subalgebras of free Lie algebras". In: *Mat. Sbornik N.S.* (1953) (cit. on p. 10).

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (cit. on p. 87).

[Stu96] Bernd Sturmfels. *Gröbner bases and Convex Polytopes*. 1996 (cit. on p. 35).

[WCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. URL: http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37 (cit. on p. 56).

[Wil12] Mark Wildon. *Dynkin-Specht-Wever Lemma*. 2012. URL: http://wildonblog.wordpress.com/2012/06/06/dynkin-specht-wever-lemma/ (cit. on p. 53).

[Wit37] Ernst Witt. "Treue Darstellung Liescher Ringe". In: *J. Reine Angew. Math.* 177 (1937), pp. 152–160 (cit. on p. 54).

[Yan+15] Weixin Yang, Lianwen Jin, Zecheng Xie, and Ziyong Feng. "Improved deep convolutional neural network for online handwritten Chinese character recognition using domain-specific knowledge". In: *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*. Aug. 2015, pp. 551–555 (cit. on p. 77).

[Yan+17] Weixin Yang, Terry Lyons, Hao Ni, Cordelia Schmid, Lianwen Jin, and Jiawei Chang. "Leveraging the Path Signature for Skeleton-based Human Action Recognition". 2017. URL: http://arxiv.org/abs/1707.03993 (cit. on p. 2).

[Yin+13] Fei Yin, Qiu-Feng Wang, Xu-Yao Zhang, and Cheng-Lin Liu. “ICDAR 2013 Chinese Handwriting Recognition Competition (report)”. 2013 (cit. on p. 77).

[Zha+18] Xu-Yao Zhang, Fei Yin, Yan Ming Zhang, Chen-Lin Liu, and Yoshua Bengio. “Drawing and Recognizing Chinese Characters with Recurrent Neural Network”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40.4 (Apr. 2018). URL: `arxiv.org/abs/1606.06539` (cit. on p. 77).

[Zie13] Günter M Ziegler. *Lectures on Polytopes*. 2013 (cit. on p. 36).