Adv. Appl. Clifford Algebras (2022) 32:24
© The Author(s) 2022
0188-7009/020001-41
published online February 22, 2022
https://doi.org/10.1007/s00006-022-01207-1

Advances in Applied Clifford Algebras

Deep Learning Gauss–Manin Connections

Kathryn Heal, Avinash Kulkarni and Emre Can Sertöz*

Abstract. The Gauss–Manin connection of a family of hypersurfaces governs the change of the period matrix along the family. This connection can be complicated even when the equations defining the family look simple. When this is the case, it is expensive to compute the period matrices of varieties in the family via homotopy continuation. We train neural networks that can quickly and reliably guess the complexity of the Gauss–Manin connection of pencils of hypersurfaces. As an application, we compute the periods of 96% of smooth quartic surfaces in projective 3-space whose defining equation is a sum of five monomials; from the periods of these quartic surfaces, we extract their Picard lattices and the endomorphism fields of their transcendental lattices.

Mathematics Subject Classification. 68T07, 32J25, 14Q10, 14C22, 32G20.

Keywords. Artificial Intelligence, K3 Surface, Neural Network, Numerical and Symbolic Computation, Period, Picard Group.

Contents

1. Introduction 3
  1.1. Application to Quartic Surfaces 3
    1.1.1. Using Periods Rigorously 4
    1.1.2. Comparison with Other Methods for Computing Picard Numbers 4
  1.2. Neural Network Heuristics for Rigorous Computations 5
  1.3. Software 5
  1.4. Main Problem: Computing Periods of Hypersurfaces 5
  1.5. Deep Learning in Algebraic Geometry 6
  1.6. Outline 7
2. Period Computation 7
  2.1. The Integral Structure 7
  2.2. The Hodge Structure on Cohomology 8

This article is part of the Topical Collection on Machine-Learning Mathematical Structures edited by Yang-Hui He, Pierre Dechant, Alexander Kasprzyk, and Andre Lukas.

*Corresponding author.

2.3. The Period Matrix 8
2.4. The Griffiths Basis for Cohomology 8
2.5. Transition Matrices for Periods 10
2.5.1. Period Transition Matrices of Linear Translates 11
2.5.2. First Order Gauss–Manin Connection 11
3. Computational Scheme 12
3.1. Two Types of Problems and a General Framework 12
3.1.1. First Problem: Computing Periods for One Target Polynomial 12
3.1.2. Second Problem: Computing Periods for Many Hypersurfaces 12
3.1.3. General Framework 13
3.2. Searching a Computation Graph for an Efficient Tree 13
3.2.1. Brute-Force Strategy 14
3.2.2. Informed Brute Force 15
4. Deep Learning Models 15
4.1. Neural Networks to Approximate Functions 16
4.2. The Class of Functions Associated to a Neural Network 16
4.3. Gradient Descent using Neural Networks 18
4.3.1. Loss Function 18
4.3.2. Gradient Descent 18
4.3.3. Stochastic Minibatch Gradient Descent 19
4.3.4. Hyperparameter Selection 19
5. A Computability Scoring Function for Estimation 19
5.1. Input Space E 20
5.2. The Subjective CSF ϕ 20
5.3. Input Augmentation 22
5.3.1. Studying the Complexity of the Input 22
5.3.2. Complexity of Cohomology Matrices 22
5.4. Dataset Preprocessing 24
5.4.1. Dimension Reduction: Principal Component Analysis 24
5.4.2. Balancing the Dataset 26
5.5. Comparison of Learning Models 26
5.5.1. Classical Statistical Methods 27
5.5.2. Deep Neural Networks 28
5.6. Implementation 29
5.6.1. Four-Monomial Quartics 29
5.6.2. Extrapolating from Four-Monomial to Five-Monomial Quartics 30
6. Application 30
6.1. Five Monomials 30
6.2. List of Results 31
6.2.1. The Missing Vertices 32
6.2.2. Isomorphism Classes 32
6.2.3. Endomorphism Fields 33
6.3. Methodology 34
6.3.1. Additional Simplifications 34

6.3.2. Computing Isomorphism Classes 34
6.4. Performance on Applications 35
6.5. Costs of Collecting the Training Data and Training the Network 36
References 39

1. Introduction

There are two ways to study deformations of algebraic varieties. One is purely *algebraic*, through the explicit polynomial equations of the family. The other is *transcendental*, through the variation of Hodge structures. The translation of the algebraic to the transcendental is achieved through the Gauss–Manin connection associated to the family. The differential equations governing the flat sections of the connection are those that trace out the variation of Hodge structures in the corresponding flag variety.

Experimentation suggests that algebraic deformations defined by simple equations can give rise to unwieldy differential equations that are well beyond our ability to integrate. This article is concerned with the following two questions: Why are these Gauss–Manin connections so complicated? How can we choose deformations with more approachable connections?

We approach these questions from a practical point of view, outlined as follows. The algorithm presented in [29] computes the variation of Hodge structures of a given pencil of hypersurfaces. A practical measure for the complexity of a Gauss–Manin connection for a pencil is the amount of time it takes for this algorithm to terminate.

We show evidence that data-driven feedforward models (e.g. an ensemble of deep neural networks) can estimate a proxy for such a practical measure of complexity, see Sects. 5.6 and 6.4. This performance can be further improved by providing a local snapshot of the Gauss–Manin connection at a few points, as in Sect. 2.5.2.

Our feedforward models outperform classical methods, see Fig. 6. We also test the neural network on datasets that are outside the scope of its training. It performs significantly better than random choice, even without training, see Figs. 10 and 11.

This paper's principal novelty is the observation that deep neural networks can capture the complexity of a Gauss–Manin connection by learning from data that are easy to compute. Furthermore, in developing a proof-of-concept calculation, we also collected a significant amount of information on quartic surfaces, which should be interesting in its own right. Naturally, we anticipate that the idea of using neural networks to optimize algorithm decisions will have a wider application in computational mathematics beyond this proof-of-concept.

1.1. Application to Quartic Surfaces

One use of computing the variation of Hodge structures is to transport the Hodge structure (i.e. periods) from one variety onto another. A high precision approximation of the period matrix of a hypersurface reveals delicate

holomorphic invariants that are very difficult to compute otherwise [22]. We will apply the techniques in *loc. cit.* to calculate the Picard numbers of the quartic surfaces considered here. This is a classical problem that has received a great deal of attention in the last two decades, see [22, 562]. Using approximate periods, one computes the Picard number with a small chance of error, as studied in *loc. cit.*

The periods allow us to compute not just the Picard number, but also the Picard lattice. Here, the Picard lattice is the Picard group together with its intersection product as well as the coordinates of a hyperplane section. This data allows one to count smooth rational curves on quartics [22, §3], for instance. Moreover, we can also check which of the quartic surfaces in our list are isomorphic by comparing their periods, and compute the endomorphism algebra of each quartic surface (Sect. 6).

We computed the periods of 96% of the smooth quartic surfaces in $\mathbb{P}^3$ that can be expressed as the sum of five monomials, each with coefficient 1. In turn, we determine their Picard lattices, their endomorphism algebras, and isomorphism classes. Although the set of quartics that we consider here appears to be small, we find 139 distinct isomorphism classes. This should be compared to the 127 isomorphism classes found amongst the 184, 000 quartics found by random search in the database produced in [22].

1.1.1. Using Periods Rigorously. The lattice and endomorphism computations depend on finding integer relations between approximately known complex numbers (the periods). The results can, therefore, be incorrect. Nevertheless, when working with high-precision approximations one expects the chance of error to be small. We leave the results in Sect. 6 in this territory of being "unproven but most-likely correct".

On the other hand, equipped with a potential answer, there are ways to prove any one of the results in Sect. 6 with some effort. By computing periods to a few thousand digits, one can expect to prove the Picard ranks computations by constructing algebraic curves representing the generators of the Picard group as in [27].

In principle, the relations between periods of quartic surfaces can be proven just by checking the relations to very high precision [23] without any additional work. However, the precision for this generic proof is unreasonably large at the moment.

Once proven, the discovered relations establish rigorous lower bounds on the Picard number. Combined with the finite characteristic techniques—see below—one can thus prove that the entire Picard lattice computation is correct, e.g. as in [27].

1.1.2. Comparison with Other Methods for Computing Picard Numbers. There is a vast literature on using finite characteristic methods to compute upper bounds on the Picard number, see the references in [6]. The most recent p-adic techniques are fast (see *loc. cit.*), and in fact, we used these methods to corroborate our Picard number computations, see Remark 6.1. However, these techniques do not reveal the Picard lattice—they only determine its rank.

Aside from a rigorous computation using the periods, the only methods that we are aware of to establish lower bounds are searching for algebraic cycles algebraically or geometrically. The former is an expensive procedure, while the latter requires human ingenuity and cannot be automated.

1.2. Neural Network Heuristics for Rigorous Computations

One of the main questions we sought to address in this article is how neural networks can be used *rigorously* for mathematical computation. Our strategy is *not* to use the network to predict what the value of the periods are for a given hypersurface. Rather, we train our neural network to recognize optimal choices within a high precision computation, namely, the algorithm of [29]. The choices in the algorithm in *loc. cit.* can be modelled as a graph traversal problem (see Sect. 3) where the cost of traversing an edge is unknown until a traversal is attempted. Because these choices have no bearing on the final result, but only on the speed of execution, we retain the reliability of the original method.

1.3. Software

The associated code is available for general use.[1] Our neural network training software is also available in this code base. In order to carry out a computation at this scale, we added a parallelization layer, data caching mechanisms, checkpointing, and fault-tolerance to the software in [29], which is used as an underlying engine.

1.4. Main Problem: Computing Periods of Hypersurfaces

The Hodge structure on a smooth hypersurface $Y = Z(g) \subset \mathbb{P}^{n+1}_{\mathbb{C}}$ of degree d can be represented by a matrix of periods $\mathcal{P}_g \subset \mathbb{C}^{m\times m}$, where m depends only on (n, d), as in Sect. 2.3. The method given in [29] of computing $\mathcal{P}_g$ involves deforming g to another smooth hypersurface $X = Z(f) \subset \mathbb{P}^{n+1}_{\mathbb{C}}$ of degree d whose periods are already known. To begin, one may take X to be a Fermat type hypersurface whose periods can be expressed by closed formulas.

Given such a pair (f, g) we will consider the pencil of hypersurfaces defined by $(1-t)f + tg$, which deforms X to Y. Explicitly representing the variation of Hodge structures from X to Y as in Sect. 2.5, one can compute (i.e. numerically approximate) a matrix $\mathcal{P}_{f,g} \in \mathbb{C}^{m\times m}$ such that if $\mathcal{P}_f$ is a period matrix of X then $\mathcal{P}_{f,g} \cdot \mathcal{P}_f$ is a period matrix of Y. We call such $\mathcal{P}_{f,g}$ a *period transition matrix*. If $Z(h) \subset \mathbb{P}^{n+1}_{\mathbb{C}}$ is another smooth hypersurface of degree d, the product of the period transition matrices $\mathcal{P}_{f,h}$ and $\mathcal{P}_{h,g}$ gives a period transition matrix from X to Y.

There is a large variation on the time to compute $\mathcal{P}_{f,g}$ in the inputs f, g. The computation of $\mathcal{P}_{f,g}$ is often time consuming, taking hours or days, but for some inputs the computation of $\mathcal{P}_{f,g}$ could take only a few seconds. A critical observation is that it is sometimes faster to compute $\mathcal{P}_{f,h}$ $\mathcal{P}_{h,g}$ than it is to compute $\mathcal{P}_{f,g}$ directly. This suggests searching for a sequence of polynomials $f{=}s_0, s_1, \ldots, s_k{=}g$ for which $\mathcal{P}_{s_i,s_{i+1}}$ is easy to compute for each

[1] The software package is available at: https://github.com/a-kulkarn/period_graph.

i. A random search based on simple heuristics was employed in [29, §3.1] to find such sequences.

Unfortunately, it is difficult to predict whether the computation of each $\mathcal{P}_{s_i,s_{i+1}}$ will terminate within k seconds without actually running the computation for k seconds. Instead, we wish to anticipate the difficulty of such a computation so that we may discard difficult pencils in favor of friendlier ones. In this article, we approach this prediction problem with deep learning.

We cast the problem of discovering a good sequence into one of finding a short path in a weighted graph. Let W be a finite set of homogeneous polynomials, all of the same degree, containing f and g. Consider the complete graph G with vertex set W and some weight function φ defined on the edges. For an edge e of G, one may define $\varphi(e)$ to be the number of seconds it takes to compute the transition matrix $\mathcal{P}_e$. Our ultimate goal is to identify a path in G that connects f and g and has small total weight. (We elaborate on how to choose W in Sect. 3.1.3.)

If the weight function φ was known, then finding such an optimal path could be solved using standard graph traversal strategies such as Dijkstra's algorithm or the A^*-algorithm [4,28]. The problem we face is that the cost of evaluating φ at an edge e is just as expensive as computing $\mathcal{P}_e$ itself. To address this, we enlist data-driven learning models to help us guess whether $\varphi(e)$ is reasonably small or not.

We train our models on a random subset E' of the edge set of the graph G. We collect information on E' by attempting a computation on each edge e in E' that is a representative fragment of the computation needed to evaluate $\mathcal{P}_e$. The models learn to recognize if, given an edge e of G, the computation $e \mapsto \mathcal{P}_e$ will terminate in a reasonable amount of time. Informed by the predictions of these models we then traverse a path in G from f to g. See Sect. 3 for more details on the general method and Sect. 5 for our implementation of the models.

Unfortunately, the edges e for which $\mathcal{P}_e$ can be readily computed are generally rare. Consequently, when k is a reasonably small threshold, the spanning subgraph of G whose edge set is $\{e \in E(G) | \varphi(e) \leq k\}$ is often sparsely connected or disconnected. One drawback of our method is that there may not exist any good path from f to g inside G, in which case we will have wasted time trying to discover one. Enlarging the polynomial set W or the threshold k may solve this problem, but at the cost of a longer training time. Further improvements may require theoretical advances into the nature of the Gauss–Manin connection. Nevertheless, what we give here significantly improves the computation time spent searching for a good connection when one exists, see Sect. 6.4.

1.5. Deep Learning in Algebraic Geometry

Deep learning can be used to find elliptic fibrations [17], to recognize isomorphism classes of groups and rings [16], and for approximating the solutions of high-dimensional partial differential equations [31]. Although there are symbolic algorithms for these tasks, they are impractical; deep learning methods

are employed to boost performance. The tradeoff for this gain in computation speed is an unreliability of the output, e.g. a natural intolerance to unavoidable approximation error.

However, directly predicting the solution to a problem is not the only way one can apply deep learning methods to mathematical computations. Instead, deep learning methods can be used to assist a more robust method by providing dynamically generated heuristics, thereby improving performance while preserving reliability—see for instance [19]. This is our approach here, see Remark 4.1.

1.6. Outline

In Sect. 2 we give an overview of the period computation strategy for hypersurfaces. In Sect. 3 we explain the problem from a computational point of view and describe the resource management strategy we employ. Section 4 gives an overview of deep learning methodology for the non-specialist. In Sect. 5 we describe our implementation of deep learning methods. In Sect. 6 we apply our code to five-monomial quartics and list their Picard numbers as well as their isomorphism classes.

2. Period Computation

Let $X = Z(f_X) \subset \mathbb{P}^{n+1}_{\mathbb{C}}$ be a smooth hypersurface where $f_X \subset \mathbb{C}[x_0, \ldots, x_{n+1}]$ is a degree d homogeneous polynomial. By the Lefschetz hyperplane theorem all cohomology groups of X are trivial (either 0 or $\mathbb{Z}$), except for the middle (singular) cohomology group $\mathrm{H}^n(X, \mathbb{Z})$.

We need to represent two kinds of structure on the cohomology groups: the integral structure and the Hodge decomposition on $\mathrm{H}^n(X, \mathbb{C})$. We recall their definition and summarize their method of computation here. For an exhaustive account, see [32,33]. The definitions that are more specific to this article are introduced in Sect. 2.5.

For a computation oriented—and, therefore, explicit—presentation of the material in this section, we refer to [29].

2.1. The Integral Structure

As an abstract group, $\mathrm{H}^n(X, \mathbb{Z})$ is isomorphic to $\mathbb{Z}^m$ for some m. After choosing an identification $\psi\colon \mathrm{H}^n(X, \mathbb{Z}) \xrightarrow{\sim} \mathbb{Z}^m$ the intersection product on $\mathrm{H}^n(X, \mathbb{Z})$ can be represented by an $m \times m$ matrix $\mathcal{I}$ with integral entries. An integral structure on $\mathrm{H}^n(X, \mathbb{C}) = \mathrm{H}^n(X, \mathbb{Z}) \otimes_{\mathbb{Z}} \mathbb{C}$ refers to an identification of the sublattice $\mathrm{H}^n(X, \mathbb{Z}) \subset \mathrm{H}^n(X, \mathbb{C})$.

Although we will suppress this from notation, whenever we refer to a trivialization ψ of the integral cohomology, we also mean a determination of the intersection product $\mathcal{I}$ on $\mathbb{Z}^m$. The integral structure on $\mathrm{H}^n(X, \mathbb{C})$ can be represented by the map $\psi_{\mathbb{C}} := \psi \otimes \mathbb{C}$.

2.2. The Hodge Structure on Cohomology

The Hodge decomposition on $\mathrm{H}^n(X,\mathbb{C}) = \mathrm{H}^n(X,\mathbb{Z}) \otimes_{\mathbb{Z}} \mathbb{C} \simeq \mathbb{C}^m$ is a direct sum decomposition:

$$\mathrm{H}^n(X,\mathbb{C}) = \bigoplus_{p=0}^{n} \mathrm{H}^{p,n-p}(X), \tag{2.1}$$

where $\mathrm{H}^{p,q}(X)$ is the space of (p,q)-forms.

The Hodge pieces $\mathrm{H}^{p,q}(X)$ do not vary holomorphically in X. Therefore, it is more natural for variational problems to consider the *Hodge filtration* $F^\ell(X) = \bigoplus_{p=\ell}^{n} \mathrm{H}^{p,n-p}(X)$ for $\ell = 0,\dots,n$. Of course, on an individual hypersurface, one can recover the decomposition from the filtration and vice versa.

Using a generic hyperplane section of X, we can define the hyperplane class $h \in \mathrm{H}^2(X,\mathbb{Z})$ in cohomology. If $n = \dim X$ is even then $h^{n/2} \in \mathrm{H}^n(X,\mathbb{Z})$ is called a polarization. In this case, the primitive part of the cohomology $\mathrm{H}^n(X,\mathsf{K})_0$ is the orthogonal complement of $h^{n/2}$, where K is any ring. If n is odd then we set $\mathrm{H}^n(X,\mathsf{K})_0 := \mathrm{H}^n(X,\mathsf{K})$. Let $m_0 := \dim_{\mathbb{C}} \mathrm{H}^n(X,\mathbb{C})_0$. The restrictions $F^\ell(X) \cap \mathrm{H}^n(X,\mathbb{C})_0$ of the Hodge filtrations to the primitive cohomology will be denoted by $F^\ell(X)_0$.

2.3. The Period Matrix

The integral structure or the Hodge structure in isolation would be discrete invariants. After all, we know $\mathrm{H}^n(X,\mathbb{Z}) \simeq \mathbb{Z}^m$ and we know the dimensions of the pieces of the Hodge decomposition (e.g. [2, §17.3]). The difficulty is putting these pieces together, which requires transcendental invariants; the periods of X.

Definition 2.1. Let us call $\mathcal{P} \in \mathbb{C}^{m\times m}$ a *period matrix* on X if there is an isomorphism $\psi\colon \mathrm{H}^n(X,\mathbb{Z}) \simeq \mathbb{Z}^m$ such that for each $\ell = 0,\dots,n$, the first $\dim_{\mathbb{C}} F^\ell(X)$ rows of $\mathcal{P}$ span $\psi_{\mathbb{C}}\left(F^\ell(X)\right) \subset \mathbb{C}^m$. Similarly, we define a *primitive period matrix* $\mathcal{P}_0 \in \mathbb{C}^{m_0\times m_0}$.

Remark 2.2. A particular matrix transformation computes the period matrix $\mathcal{P}$ from the primitive period matrix $\mathcal{P}_0$, and vice versa. See [22, §7] for the determination of this matrix transformation, which depends only on the degree and dimension of the hypersurface.

2.4. The Griffiths Basis for Cohomology

Different identifications of $\mathrm{H}^n(X,\mathbb{Z})$ with $\mathbb{Z}^m$ will change $\mathcal{P}$ by the action of the discrete group $\mathrm{GL}(m,\mathbb{Z})$. However, there is a continuous family of choices to be made in choosing a basis for $F^\ell(X)$. We now eliminate this indeterminacy by specifying a construction for a well-defined basis for cohomology compatible with the filtration $F^\ell(X)$. In particular, through these bases, the period transition matrices of Sect. 2.5 become uniquely defined.

Let $S = \mathbb{C}[x_0,\dots,x_{n+1}]$, $\mathrm{Jac}(f_X) = (\partial_0 f_X,\dots,\partial_{n+1} f_X)$ be the Jacobian ideal and $R = S/\mathrm{Jac}(f_X)$. Since X is smooth, R is a finite dimensional algebra over $\mathbb{C}$. Let us write R_ℓ for the quotient of the homogeneous part $S_\ell/\mathrm{Jac}(f_X)_\ell$.

For each $\ell \geq 0$, Griffiths [14,15] defines a *residue map*:

$$\mathrm{Res}\colon S_{(n+1-\ell)d-n-2} \to F^{\ell}(X)_0 : p \mapsto \mathrm{res}\, \frac{p}{f^{n+1-\ell}} \mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}, \tag{2.2}$$

where $\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}$ is the natural generator of the twisted canonical bundle $\Omega^{n+1}_{\mathbb{P}^{n+1}/\mathbb{C}}(n+2)$ and is given by

$$\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}} := \sum_{i=0}^{n+1} (-1)^i x_i \mathrm{d}x_0 \wedge \cdots \wedge \widehat{\mathrm{d}x_i} \wedge \cdots \wedge \mathrm{d}x_{n+1}. \tag{2.3}$$

For an $(n+1)$-form η on $\mathbb{P}^{n+1}$ with pole on X, and an n-cycle $\eta \in \mathrm{H}_n(X,\mathbb{Z})$ one computes the pairing of γ and $\mathrm{res}\,\eta$ by integrating η on a small S^1-bundle around γ lying in the complement of X [14].

These residue maps descend to an isomorphism on the quotients:

$$R_{(n+1-\ell)d-n-2} \xrightarrow{\sim} F^{\ell}(X)_0/F^{\ell+1}(X)_0, \quad \forall \ell = 0, \ldots, n.$$

Impose the grevlex ordering on S and consider the ideal of leading terms $\mathrm{lt}(\mathrm{Jac}(f_X))$ of $\mathrm{Jac}(f_X)$. The grevlex ordering gives a well defined sequence of monomials which descend to a basis of $S/\mathrm{lt}(\mathrm{Jac}(f_X))$, and therefore to a basis of R. Their residues $\omega_1, \ldots, \omega_{m'}$ in appropriate degrees yield a basis of the primitive cohomology $\mathrm{H}^n(X,\mathbb{C})_0$.

Definition 2.3. The basis $\omega_1, \ldots, \omega_{m'}$ of the primitive cohomology constructed above is a well defined basis which respects the filtration. We will call this basis *the (grevlex) Griffiths basis* on X.

Definition 2.4. A primitive period matrix $\mathcal{P}_0$ as in Definition 2.1 will be called a *primitive grevlex period matrix* of $X = Z(f_X)$ if the i-th row of $\mathcal{P}_0$ equals $\psi_{0,\mathbb{C}}(\omega_i)$, where $\psi_0\colon \mathrm{H}^n(X,\mathbb{Z})_0 \xrightarrow{\sim} \mathbb{Z}^{m_0}$ and $\{\omega_j\}_{j=1}^{m_0}$ is the grevlex Griffiths basis on X. Any period matrix obtained by extending $\mathcal{P}_0$ as in Remark 2.2 will be called a *grevlex period matrix* of X.

Example 2.5. Consider the Fermat quartic $X = Z(f) \subset \mathbb{P}^4$, with $f = x^4 + y^4 + z^4 + w^4$. The primitive grevlex Griffiths basis for X is given by the differentials $\omega_1, \ldots, \omega_{21}$, which are the residues of:

$$\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f},$$

$$z^2w^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ yzw^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ xzw^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ y^2w^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},$$

$$xyw^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ x^2w^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ yz^2w\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ xz^2w\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ y^2zw\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},$$

$$xyzw\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ x^2zw\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ xy^2w\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ x^2yw\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ y^2z^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},$$

$$xyz^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ x^2z^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ xy^2z\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ x^2yz\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},\ x^2y^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^2},$$

$$x^2y^2z^2w^2\frac{\mathrm{vol}_{\mathbb{P}^{n+1}_{\mathbb{C}}}}{f^3}.$$

Upon choosing some identification $\psi_0\colon \mathrm{H}^n(X,\mathbb{Z})_0 \xrightarrow{\sim} \mathbb{Z}^{m_0}$ (viewed as a row vector), the grevlex primitive period matrix of X is the square matrix

$$\mathcal{P}_0 = \begin{bmatrix} \psi_0(\omega_1) \\ \vdots \\ \psi_0(\omega_{21}) \end{bmatrix}.$$

Any two grevlex period matrices of X differ by the action of the discrete group $\mathrm{GL}(m,\mathbb{Z})$. Moreover, after fixing a coordinate system on the integral primitive cohomology, the grevlex period matrix is uniquely defined. The choice of convention made here is purely to ensure that period transition matrices discussed in the following section are well-defined objects.

2.5. Transition Matrices for Periods

Consider a pair of smooth hypersurfaces $X = Z(f)$ and $Y = Z(g)$ with $f, g \in \mathbb{C}[x_0,\dots,x_{n+1}]_d$.

Definition 2.6. We will call a matrix $\mathcal{P}_{f,g} \in \mathbb{C}^{m_0\times m_0}$ a *period transition matrix* if for any grevlex primitive period matrix $\mathcal{P}_f$ on $X = Z(f)$, the product $\mathcal{P}_{f,g}\cdot\mathcal{P}_f$ is a primitive grevlex period matrix on $Y = Z(g)$.

A method for computing a period transition matrix $\mathcal{P}_{f,g}$ is explained in [29]. A much simpler method is available when g is a linear translate of f and we will do this example in Sect. 2.5.1. We will briefly outline the general method of [29] here.

1. Take a one parameter family of hypersurfaces $\mathcal{X}_t = Z(f_t)$ where $f_t \in \mathbb{C}(t)[x_0,\dots,x_{n+1}]_d$ with $X = \mathcal{X}_0$ and $Y = \mathcal{X}_1$.
2. Find polynomials $p_1,\dots,p_{m'} \in \mathbb{C}(t)[x_0,\dots,x_n]$ which descend to a basis for $\bigoplus_{\ell=0}^n \mathbb{C}(t)[x_0,\dots,x_n]_{(n+1-\ell)d-n-2}/\operatorname{Jac}(f_t)_{(n+1-\ell)d-n-2}$ and also to bases when $t=0$ and $t=1$. Often p_i are monomials or a sum of two monomials (with constant coefficient 1).
3. Find the matrix B expressing the change of basis from the basis above to the grevlex basis at $t=1$. This is done by the computation of normal forms.
4. For each $i = 1,\dots,m'$ find a differential operator $\mathcal{D}_i \in \mathbb{C}(t)[\frac{\partial}{\partial t}]$ such that $\mathcal{D}_i\cdot\operatorname{Res}(p_i) = 0$. These differential equations annihilate the rows of the grevlex period matrix of $\mathcal{X}_t$.
5. Find a path γ from 0 to 1 in $\mathbb{C}\setminus\mathcal{S}$ where $\mathcal{S}$ is the set of values of t for which $\mathcal{X}_t$ is singular.
6. Compute the transition matrix for the space of solutions of $\mathcal{D}_i$ at $t=0$ and $t=1$ obtained by homotopy continuation along γ.
7. Using the indicial equation of $\mathcal{D}_i$, find the set of derivatives of $\operatorname{Res}(p_i)$ whose values at $t=0$ would determine $\operatorname{Res}(p_i)$.
8. Express the derivatives of $\operatorname{Res}(p_i)$ at $t=0$ in terms of the grevlex Griffiths basis at $t=0$. Multiply this expression on the right with the transition matrix of (2.5) and take the first row.
9. Form the $m'\times m'$ matrix whose i-th row is the row obtained at the step above and multiply it by B on the left to get $\mathcal{P}_{0,1}$.

The most time expensive step is step (2.5). (The complexity of step (2.5) depends on the γ chosen in step (2.5); fortunately, there is a heuristic to choose γ optimally—see [29].) However, most attempts to compute the period transition matrix fail on step (2.5). This is because it requires a Gröbner basis computation for $\mathrm{Jac}(f_t)$, numerous normal form computations, and expressions of elements in $\mathrm{Jac}(f_t)$ in terms of the generators $(\partial_0 f_t, \ldots, \partial_{n+1} f_t)$. We provide statistical evidence for the claim regarding step 2.5 on our main dataset in Sect. 5.2, see the paragraph before Definition 5.1.

2.5.1. Period Transition Matrices of Linear Translates. If two hypersurfaces are linear translates of one another, then a period transition matrix between them can be easily computed. We will do this here as an instructive example. This is also a computation we will use later in the article in Sect. 6.

The general linear group $\mathrm{GL}(n+2, \mathbb{C})$ acts on the coordinates on $\mathbb{P}^{n+1}$ linearly. The induced action on the coordinate ring $S = \mathbb{C}[x_0, \ldots, x_{n+1}]$ is given by $u \cdot f(x) = f(x \cdot u^t)$ where $u \in \mathrm{GL}(n+2, \mathbb{C})$, $f \in S$, u^t is the transpose of the matrix u, and $x = (x_0, \ldots, x_{n+1})$ is treated as a row vector.

With $X = Z(f)$ a smooth hypersurface as before, suppose that $Y = Z(g)$ with $g = u \cdot f$ for some $u \in \mathrm{GL}(n+2, \mathbb{C})$. If $\mathcal{P}_X$ is a period matrix of X then it certainly works as a period matrix for Y. However, even if $\mathcal{P}_X$ is a grevlex period matrix on X it need not be grevlex on Y. We describe the computation of a period transition matrix $\mathcal{P}_{f,g}$ below.

Let $p_1, \ldots, p_{m'} \in S$ be the polynomials whose residues give the grevlex Griffiths basis on X. Let $\phi \colon Y \xrightarrow{\sim} X$ be the isomorphism induced by u. If $\mathcal{P}_X$ is a primitive grevlex period matrix on X then $\mathcal{P}_X$ is a primitive period matrix on Y whose rows represent the residues of $u \cdot p_1, \ldots, u \cdot p_{m'}$.

We can use Griffiths–Dwork reduction (see [14,15] or a summary [29, §2.4]) on Y to write each $\mathrm{Res}(u \cdot p_i)$ in terms of the grevlex Griffith basis on Y. If N is a matrix whose rows store the coordinates of $\mathrm{Res}(u \cdot p_i)$ in the grevlex basis then $N^{-1} \cdot \mathcal{P}_X$ will be a primitive grevlex period matrix on Y. In other words, N^{-1} is a period translation matrix from X to Y. This computation is implemented in `PeriodSuite`[2] as the function `translate_period_matrix`.

2.5.2. First Order Gauss–Manin Connection. We would like to detect which of the differential operators $\mathcal{D}_i$ appearing in Item (2.5) of Sect. 2.5 would be easy to integrate, before we compute $\mathcal{D}_i$.

Collectively, these $\mathcal{D}_i$ define flat sections of the Gauss–Manin connection of the family $\mathcal{X}_t$ (see for example [32, §9.3]). Computing the Gauss–Manin connection itself is simpler than computing its flat sections, but still not quite fast enough for rapid testing. Computing the Gauss–Manin connection evaluated at a single point, however, is very fast. Furthermore, these evaluations give an impression of how complicated $\mathcal{D}_i$ might be (see Sect. 5.5).

Let $\{\omega_i = \mathrm{Res}(p_i)\}_{i=1}^{m'}$ be a basis for primitive cohomology on a hypersurface X. Say X is the fiber of a family $\mathcal{X}_t$ at $t = t_0$. Then the residue of the polynomials p_i on the family $\mathcal{X}_t$ will give a basis for primitive cohomology for all t near $t = t_0$. Differentiating these forms with respect to t and evaluating

[2] https://github.com/emresertoz/PeriodSuite.

at $t = t_0$ gives new elements in the cohomology of X. Expressing these new forms in terms of $\{\omega_i\}_{i=1}^{m'}$ gives an $m' \times m'$ matrix. We call this matrix the *first order Gauss–Manin matrix*.

Let us note that if X and the polynomials p_i are defined over a subfield $K \subset \mathbb{C}$ then the corresponding first order Gauss–Manin matrix will have entries in K. See [20] for more on this topic. In our applications, we will take $K = \mathbb{Q}$.

3. Computational Scheme

We will now describe our approach to the problem of searching for a good path between polynomials, with the goal of transferring their periods from one to the other. There are two variations of this problem that we are interested in solving. We will state and explain these variations and then generalize them to a common framework in Sect. 3.1. In Sect. 3.2 we will explain how we operate in this abstract framework.

3.1. Two Types of Problems and a General Framework

We are interested in solving two types of problems. In the first problem, we are given a pair of polynomials $V = \{f, g\}$ depicting smooth hypersurfaces of the same degree, and our goal is to compute the period transition matrix $\mathcal{P}_{f,g}$. In the second problem, we are given a (possibly large) set of polynomials V, and our goal is to compute the periods of all elements in V given the periods of any one of them. We will now present some example problems that we will solve later on.

3.1.1. First Problem: Computing Periods for One Target Polynomial. For a general (f, g), a direct computation of $\mathcal{P}_{f,g}$ is infeasible even with low precision. The path from f to g must be broken into simpler pieces. The strategy we discussed in Sect. 1.4 is to find a sequence of polynomials $f = h_0, h_1, \dots, h_s = g$ such that each intermediate period transition matrix $\mathcal{P}_{h_i,h_{i+1}}$ is easily computable. Their product would then give $\mathcal{P}_{f,g}$. This problem was already investigated in [29, §3.1] and a crude heuristic was developed there. We will develop this heuristic further in this section.

3.1.2. Second Problem: Computing Periods for Many Hypersurfaces. Here we are given a (possibly large) set of polynomials V and we would like to be able to compute the periods of all elements in V given the periods of any one of them. For example, the set V may be the set V_n consisting of all smooth quaternary quartics that are expressed as the sum of n distinct monomials with coefficients equal to 1, e.g.

$$V_4 = \{x^4 + y^4 + z^4 + w^4,\ x^3y + xy^3 + z^3w + w^4, \dots \}.$$

The set V_4 has 108 elements, and V_5 has 3348 elements. There is a natural action of S_4 on these sets given by permuting the variables, and any two polynomials related by an element of S_4 define isomorphic quartic hypersurfaces. We have that $\#(V_4/S_4) = 10$ and $\#(V_5/S_4) = 161$.

An attempt to compute the periods, and therefore the Picard numbers, of all elements in V_5 was made in [22]. Many of the elements in V_5 were out of reach at the time. We apply our methods to compute the periods of most elements in V_5, see Sect. 6.

Given a sequence of polynomials $f = h_0, h_1, \dots, h_s = g$ such that each intermediate period transition matrix $\mathcal{P}_{h_i,h_{i+1}}$ is easily computable, and such that the periods of f are known to some precision, we easily obtain the periods of each intermediate quartic by the partial products $\mathcal{P}_{h_i,h_{i+1}} \dots \mathcal{P}_{h_0,h_1}\mathcal{P}_f$. Thus, it suffices to determine a set of paths which connect the vertices of V as opposed to computing the periods one by one.

3.1.3. General Framework. Both problems can be slightly generalized to fit into the following framework. Suppose we are given a set V containing one element whose periods are known and the periods of all the others are sought.

The set V may not have enough pairs $f, g \in V$ such that $\mathcal{P}_{f,g}$ is directly computable. In this case, we need to construct a larger set W containing V which introduces many pairs $(f, g) \in W \times W$ such that $\mathcal{P}_{f,g}$ is directly computable. The construction of this W can be based on human heuristics.

Letting K_W be the complete graph with vertex set W, we now wish to solve the following problem: *Given $V \subset W$, find a tree $T \subset K_W$ such that the vertex set of T contains V and the computation of the period transition matrix for each edge in T is feasible.* Constructing W is a balancing act. If the search space W is too large, it may be impractical to *find* a good tree T inside K_W, even though one *may exist*. Conversely, if W is too small, we may be able to search the entire space but find that there is no good tree T inside K_W.

For our first type of problem (Sect. 3.1.1), with $V = \{f, g\}$, one may take the following set:

$$W = \{h \mid \operatorname{supp}(h) \subset \operatorname{supp}(f) \cup \operatorname{supp}(g),\ \operatorname{coefs}(h) \subset \operatorname{coefs}(f) \cup \operatorname{coefs}(g)\}$$

where supp denotes the monomial support of a polynomial and coefs denotes the set of coefficients of a polynomial. For the second type of problem we will consider $V = V_5$. In this case, it seems natural to take something in between $W = V_4 \cup V_5$ and $W = V_4 \cup V_5 \cup V_6$.

Remark 3.1. There are numerous modifications to this problem that can help attain the final goal. We will only point out one: it is often sensible to prune the graph K_W by a problem-specific heuristic before embarking on a search for tree T.

3.2. Searching a Computation Graph for an Efficient Tree

In this section, we will set aside our discussion of period computations. We are interested only in abstracting the problem laid out in Sect. 3.1.3 to, essentially, a problem in resource management in a computational exploration.

Let $G = (W, E)$ be a graph with vertex set W and edge set E. Consider a computationally expensive program $\mathcal{P}$ which takes as input an edge $e \in E$ and returns an output $\mathcal{P}_e$. We will consider a cost function $\varphi\colon E \to \mathbb{R}_+$ that measures the difficulty of performing the computation $e \mapsto \mathcal{P}_e$. Common

examples of $\varphi(e)$ would be the number of arithmetic operations needed to complete the program $e \mapsto \mathcal{P}_e$ or the amount of memory required to execute the program. For our purposes, it is important that a lower bound for φ can be determined in finite time. Henceforth, we fix one such φ and refer to it as a *complexity function.*

Remark 3.2. In the context of this paper, W is a set of polynomials. For each edge e, our procedure outputs a period transition matrix $\mathcal{P}_e$ to a fixed degree of precision. The value of $\varphi(e)$ may then be the number of seconds it took to perform the computation $e \mapsto \mathcal{P}_e$ on a fixed computer. These specifics will not be relevant for the rest of this section, as we continue with the abstraction above.

One could rephrase the problem introduced in the previous section as follows: *Given $V \subset W$, find a tree $T \subset G$ such that V is contained in the vertex set of T and $\varphi(T) = \sum_{e \in T} \varphi(e)$ is small, if not minimal.*

If we know the value of φ on each edge, then this problem becomes a standard minimization problem. However, at the outset, the evaluation of φ is just as expensive as the computation $e \mapsto \mathcal{P}_e$ itself. This leads to the following brute-force strategy.

3.2.1. Brute-Force Strategy. The brute force method (Algorithm 1) attempts to compute every edge in the graph until the desired tree is constructed. If the computation of an edge takes too long, then the computation is aborted and the computation for the next edge is begun.

Algorithm 1 Brute force with thresholding

Input: V—Set of target vertices
  W—Set of waypoint vertices containing V
  E—List of edges to be attempted
  k—Threshold parameter, e.g. maximum time per attempt
  Q—The job queue, i.e. a bijection $Q\colon \{1, \ldots, \#E\} \to E$

1: Let $G' = (W, \emptyset)$ be the graph on W with empty edge set.
2: **while** $n \leq \#E$ **do**
3:   **Attempt** the computation $n \mapsto \mathcal{P}_{Q(n)}$.
   Abort if $\varphi(Q(n)) \geq k$ is detected.
4:   **if** successful **then**
5:     Set $G' := G' \cup \{Q(n)\}$
6:     **if** V is contained in a connected component of G' **then**
7:       **return** G'
8:   Increment n
9: **return** Fail
  {(One may apply multiple rounds with enlarged W and k to eventually succeed.)}

In practice, we perform the computations in the job queue in parallel, but the size of the edge set ($\approx 10^6$) is so much larger than the number of

cores available to us ($\approx 10^2$) that this serial conceptualization is not too far off the mark.

3.2.2. Informed Brute Force. It is clear that the choice of the ordering Q on the edge set will make a dramatic impact on the total time it takes to find T. In ideal cases, the first few edges in the sequence might be easily computable and sufficient to form a usable tree T. In this case, we could stop searching early. In unfortunate cases, all of the edges e for which $\varphi(e) > k$ might be queued up first in the list, and all of our time would be spent trying to compute impossible edges.

For this reason, we consider finding a good edge ordering Q our top priority. Equivalently, our task is to find an efficiently computable weight function $\phi\colon E \to [0, 1]$ for which sorting the edges by weight gives a favorable ordering. In Sect. 5, we discuss how to use a neural model as such a function. As we perform computations, we can refine our weight function in order to improve the reliability. This suggests the modification of the brute force algorithm given by Algorithm 2.

Algorithm 2 Modified brute force

Input: V—Set of target vertices
W—Set of waypoint vertices containing V
E—List of edges to be attempted
k—Threshold parameter. (Maximum time per attempt, in seconds.)
ϕ—[Optional] Heuristic approximation to the cost function

1: **if** ϕ is provided **then**
2: Set $\leq(e, f) := (\phi(e) \leq \phi(f))$ (function returning a Boolean)
3: Set $Q :=$ `sort_descending_order`$(E, \leq)$
4: **else**
5: Set $Q :=$ `randomize_order`(E)
6: Set result $:=$ `BruteForce`(E, k, Q).
Collect data during this computation in order to retrain the models and improve the reliability of $\phi(e)$.
7: **if** result is not Fail **then**
8: **return** result
9: Possibly enlarge W or increase k.
10: Retrain ϕ.
11: **goto** 1, using the improved ϕ.

4. Deep Learning Models

In this section we provide a brief overview of neural networks for the mathematician who does not specialize in applied or computational mathematics. An excellent reference for this subject is [13].

4.1. Neural Networks to Approximate Functions

A neural network is essentially a framework for approximating functions. Let $\varphi \colon \mathbb{R}^n \to \mathbb{R}^m$ be an unknown "targe" function that we wish to approximate, and let $\Gamma_\varphi \subset \mathbb{R}^{n+m}$ be the graph of φ. Suppose that we have access to a finite subset $\mathcal{T}$ sampled from Γ_φ. Assuming strong hypotheses on $\mathcal{T}$ and φ, we can in some cases closely approximate or even recover φ.

A classical example of such a scenario is univariate polynomial interpolation. If the target function φ is known to be a polynomial with degree bounded above by some known k, then we can closely approximate φ from $\mathcal{T}$ (provided $\mathcal{T}$ is sampled sufficiently well) by solving a reasonably-sized linear system. This example demonstrates a good use-case for polynomial interpolation. However, the linear system could be defined by a Vandermonde matrix that is extremely ill-conditioned, making a solution very difficult and sensitive to noise.

Thus interpolation may not always be the most appropriate way to learn φ. In many real-world scenarios (e.g. that encountered in this work), we cannot assume that φ is polynomial. A function φ that is *not* sufficiently regular may require interpolating polynomials of very high degree. This can result in a high-dimensional linear system requiring a large matrix inversion, which can be a computationally prohibitive task. In this work, we relax our requirement of a perfect fit between model and data, and use a neural network architecture with a variation of gradient descent to solve such a regression problem.

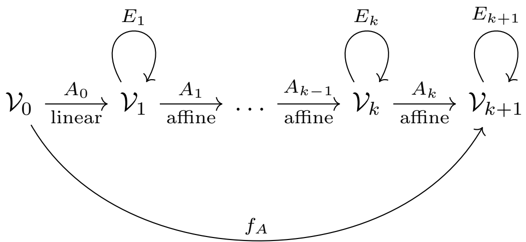
In mathematical analysis, it is common practice to construct a class of functions $\mathcal{C}$ which exhibits so-called good approximating properties. Once an appropriate class has been chosen, one seeks a sequence of functions $\{\varphi_n\}_{n=1}^{\infty} \subset \mathcal{C}$ which converges in some sense to φ. As will be discussed below, many chosen neural network architectures determine such a class $\mathcal{C}$ [8,12,18,25] and the training regime provides an algorithm for constructing an approximating sequence $\{\varphi_n\}_{n=1}^{\infty}$ using $\mathcal{T} \subset \Gamma_\varphi$ that will hopefully converge to φ. Note that the class $\mathcal{C}$ is by no means guaranteed to be unique or best for the task at hand. This framework is more generally referred to in computer science as *regression analysis*.

Remark 4.1. We seek to approximate a complexity function (as in Sect. 3.2) associated to the period computations, rather than approximating the period function itself. This is because our intended use for the periods require hundreds of digits of accuracy and, ideally, rigorous bounds on error. Approximating functions with neural networks typically capture large scale features of a function. Attaining high precision, let alone bounding the error, is not one of the highest priorities of the method. Furthermore, we can better tolerate error in the complexity function as we will use it only as a heuristic to order our computations (Sect. 3.2.2).

4.2. The Class of Functions Associated to a Neural Network

We define a *neural network* as a triplet $\mathcal{N} = (\mathcal{V}, E, A)$ where $\mathcal{V} = (\mathcal{V}_0, \ldots, \mathcal{V}_{k+1})$ is a sequence of real vector spaces with a fixed coordinate system $\mathcal{V}_i = \mathbb{R}^{n_i}$,

$E = (E_1, \ldots, E_{k+1})$ is a sequence of non-linear transformations $E_i : \mathcal{V}_i \to \mathcal{V}_i$ and $A = (A_0, \ldots, A_k)$ is a sequence of *affine* transformations $A_i : \mathcal{V}_i \to \mathcal{V}_{i+1}$. By an affine transformation we mean the composition of a linear map $\mathcal{V}_i \to \mathcal{V}_{i+1}$ with a translation $\mathcal{V}_{i+1} \to \mathcal{V}_{i+1}$.



The non-linear transformations E_i are typically chosen to be of a very specific form in order to facilitate computations and to enable gradient propagation. They do not all need to be the same function.

Definition 4.2. For any $x \in \mathbb{R}$ the function $x^+ = \max(0, x)$ is called the *rectifier*. For $W = \mathbb{R}^n$, the *rectified linear unit (ReLU) on* W is the non-linear map $W \to W$ defined by $(x_1, \ldots, x_n) \mapsto (x_1^+, \ldots, x_n^+)$.

Unless specified otherwise, we will always take each E_i to be a ReLU.

The *architecture of a neural network* consists of the choice of $\mathcal{V}$ and E. During the training of the neural networks, the architecture remains fixed and only the affine transformations A are changed. The entries of the matrices representing A_i's are called *parameters* of the neural network $\mathcal{N}$.

When the architecture $(\mathcal{V}, E)$ is fixed, we suppress it from notation and associate to each neural network $(\mathcal{V}, E, A)$ the function $f_A : \mathcal{V}_0 \to \mathcal{V}_{k+1}$ defined as follows:

$$f_A : v \mapsto \underbrace{E_{k+1} \circ A_k}_{\text{output layer}} \circ E_{k-1} \circ \cdots \circ A_1 \circ \underbrace{E_1 \circ A_0}_{\text{input layer}} (v) \tag{4.1}$$

for some fixed k of choice. Networks are defined in part by their *hyperparameters*, i.e. certain characteristics (e.g. the *depth* k of the network and the *width* $\dim(\mathcal{V}_i)$ of each layer) that define the class $\mathcal{C}$ of functions allowed by the neural network. We will call each $A_i \circ E_{i+1}$ a *layer* of the neural network $\mathcal{N}$. We refer to layer $i = 0$ (resp. $i = k$) as the *input* (resp. *output*) *layers*; the remaining layers are called *hidden layers*. The naming suggests the layers' use and the asymmetry apparent in the construction of f_A. The affine transformations A_i are parametrized by variables called *weights* and variables called *biases*. In this setting we will equate a neural network simply with a composition of such layers.

It is remarkable that something as simple as the incorporation of ReLU functions vastly expands the space of functions that can be represented by neural networks. Without the non-linearity of the transformations E, the function f_A would simply be an affine transformation.

4.3. Gradient Descent using Neural Networks

Fixing the architecture $(\mathcal{V}, E)$ of a neural network yields a parametrized family of neural networks $(\mathcal{V}, E, A)$, which can be associated to the family of functions $\mathcal{C} = \{f_A|A\}$. For each $i > 0$, the affine transformation A_i has $n_i n_{i+1} + n_{i+1}$ trainable parameters, so the total number of parameters in f_A is

$$N = \sum_{i=0}^{k} (n_i + 1)n_{i+1}. \tag{4.2}$$

That is, we have a parametrization $\mathbb{R}^N \twoheadrightarrow \mathcal{C} = \{f_A|A\}$. A choice of transformation $A^{(0)} \in \mathbb{R}^N$ induces a neural network function $f_{A^{(0)}} : \mathcal{V}_0 \to \mathcal{V}_{k+1}$.

Our immediate goal is to find a sequence $A^{(k)} \in \mathbb{R}^N$, $k \geq 0$, such that the sequence of functions $\{f_{A^{(k)}}\}_{k=0}^{\infty}$ converges in a sense to an approximation of our target function $\varphi\colon \mathcal{V}_0 \to \mathcal{V}_{k+1}$. We will first describe the distance measure that defines convergence for our experiments.

4.3.1. Loss Function. Given a function $g\colon \mathcal{V}_0 \to \mathcal{V}_{k+1}$ we wish to quantify the goodness of our network's function approximation, i.e. how far g is from being equal to φ. The only information we are given about φ is the finite subset $\mathcal{T} \subset \Gamma_\varphi \subset \mathcal{V}_0 \times \mathcal{V}_{k+1}$ of its graph; this subset is called the network's *training set*. As we have a fixed coordinate system on $\mathcal{V}_{k+1}$, we will use the induced Euclidean norm $\|\cdot\|\colon \mathcal{V}_{k+1} \to \mathbb{R}_{\geq 0}$.

Let $P(\Gamma_\varphi)$ denote the set of finite subsets of Γ_φ. Let $\mathcal{L}\colon \mathbb{R}^N \times P(\Gamma_\varphi) \to \mathbb{R}_{\geq 0}$ be the *loss function* defined as follows:

$$\mathcal{L}(A; \mathcal{T}) := \sum_{(t_1,t_2)\in\mathcal{T}} \|f_A(t_1) - t_2\|^2. \tag{4.3}$$

As in the classical philosophy of regression, we say that *to fit φ well given $\mathcal{T}$ is to minimize $\mathcal{L}$ with respect to A*. In the following we discuss two variations of a popular iterative method with the aim of achieving the minimization

$$\arg\min_{A\in\mathbb{R}^N} \mathcal{L}(A; \mathcal{T}).$$

The following class of algorithms provides a sequence $\{f_{A^{(k)}}\}_{k=0}^{\infty}$ whose $\mathcal{L}$-values will hopefully (and in some limited cases, provably) be decreasing.

4.3.2. Gradient Descent. Given $\mathcal{T} \subset \Gamma_\varphi$ we wish to find $A \in \mathbb{R}^N$ minimizing the error function $\mathcal{L}(\cdot, \mathcal{T})$. Our restriction to feedforward networks forbids feedback loops and, therefore, allows for an easy evaluation of the gradient $\nabla\mathcal{L}(\cdot, \mathcal{T})$ of $\mathcal{L}(\cdot, \mathcal{T})$ at any given point A via backpropagation—see [13] for definitions and details.

Choose an initial point $A^{(0)} \in \mathbb{R}^N$ and a sequence of *step sizes* $\gamma\colon \mathbb{N} \to \mathbb{R}_{>0}$. This sequence is typically either constant, or converging to zero. Inductively define the following sequence:

$$A^{(k)} := A^{(k-1)} - \gamma(k)\nabla\mathcal{L}(A^{(k-1)}; \mathcal{T}), \quad k > 0. \tag{4.4}$$

In the applications we have in mind, the size of $\mathcal{T}$ will be too large to make the execution of this method feasible. In examples where $\#\mathcal{T}$ is

large, one might opt for some variant of *stochastic gradient descent*; one such method is described below.

4.3.3. Stochastic Minibatch Gradient Descent. Stochastic minibatch gradient descent differs from gradient descent in that one trains on a random subset of $\mathcal{T}$ at each step, instead of $\mathcal{T}$ itself. To do this, fix a *batch size* $b \in \mathbb{N}$, and define $A^{(k)}$ inductively as follows:

Choose a random subset $\mathcal{T}_k \subset \mathcal{T}$ of size b and let

$$A^{(k)} := A^{(k-1)} - \gamma(k)\nabla\mathcal{L}(A^{(k-1)}; \mathcal{T}_k), \quad k > 0. \tag{4.5}$$

4.3.4. Hyperparameter Selection. In order to use a neural network architecture $(\mathcal{V}, E)$ to approximate the function φ, for which a subset $\mathcal{T} \subset \Gamma_\varphi$ is known, one requires the following: An error function $\mathcal{L}$, step sizes γ, batch sizes b, distributions to randomly choose subsets of $\mathcal{T}$ and to choose a starting point $A^{(0)}$. These choices $(\mathcal{V}, E, \mathcal{L}, \gamma, b)$ are the hyperparameters of the network. This defines the class $\mathcal{C}$ as described in Sect. 4.1.

Selecting hyperparameters can be more of an art than a science. Although principled selection strategies have been proposed ([5] provides an excellent survey of some such methods), this problem remains largely unsolved for general learning problems. The rate of convergence of an iterative algorithm, and even whether the algorithm converges at all, can depend heavily on parameter choice. For example, a step size chosen too small will cause the algorithm to crawl slowly to a local minimum, whereas a step size that is too large might cause the algorithm to diverge. Experimentation is required to select hyperparameters in a way so that the stochastic gradient converges rapidly and to a reasonable approximation of φ.

5. A Computability Scoring Function for Estimation

Section 3.2 introduced two problems related to the traversal of a graph whose weights are partially unavailable to us. In the current section, we explore various methods to *learn a computability scoring function (CSF)* ϕ *that is a reasonable proxy for the complexity function* φ. The design space for our experiments includes a choice of labelled training dataset $\mathcal{T}$ and a statistical model (classical or deep neural).

Toward our comparison of statistical models, we refer the reader to Sect. 4 for a brief introduction to deep neural networks. In that section, we offer definitions and outline standard training procedures. In the current section we will describe our implementation of such deep learning models (e.g. hyperparameter selection) for the computational problems at hand. Sections 5.1 and 5.2 are concerned with the design of the dataset on which we will learn. Section 5.3 compares useful intermediate representations of this dataset, a subproblem known as *feature extraction*. Section 5.6 discusses the experiments with 4-and 5-monomial quartic surfaces that motivated our choice of hyperparameters.

5.1. Input Space E

Assuming the kind of function ϕ we seek is sufficiently smooth, we can approximate it via statistical learning methods using only finitely many pairs $(e, \phi(e))$. We will choose a random subset $E' \subset E$ and assign a value $\phi(e)$ for each $e \in E'$. The resulting set of pairs

$$\mathcal{T} = \{(e, \phi(e)) | e \in E'\} \tag{5.1}$$

will be used to train a statistical model to obtain a modest guess for what ϕ should be. The remaining pairs $E \backslash E'$ will be used for testing and validation of the trained model.

The abstract discussion in this chapter applies in the full generality of Sect. 3.1. Recall the notation of Sect. 3.1.2, where $V_k \subset \mathbb{Q}[x, y, z, w]_4$ denotes the *k-nomial data set*, that is, the set of four-variable homogeneous polynomials that are the sum of k distinct monomials all with coefficient 1. For this section, we will constrain ourselves to fixed sets of polynomials—i.e. to the complete graphs on V_4 and V_5. We denote by E the edges of the graph V_k that is eventually to be traversed. For instance, the 4-nomial data set is defined by the complete graph on V_4 and thus $\#E = \binom{V_4}{2}$.

5.2. The Subjective CSF ϕ

Since we are interested in sorting edges by complexity, we actually want to learn a weight function $\phi\colon E \to [0, 1]$ which assigns to each edge $e \in E$ a probability that the complexity $\varphi(e)$ of the computation $e \mapsto \mathcal{P}_e$ is low. One could, in principle, define the weighting function $\phi = \frac{1}{1+\varphi}$ where φ is the complexity function introduced in Sect. 3.2. However, the difficulty in explicitly computing φ motivates us to find a more practical, data-driven solution.

While the complexity $\varphi(e)$ reflects some objectively defined feature of an edge, namely the number of operations and amount of memory used by a particular algorithm with input e, in practice we can only glimpse $\varphi(e)$ through the subjective lens of running the algorithm on a specific piece of hardware. From an engineering standpoint, this is the only relevant measurement of complexity if one is principally concerned with the results of the computation (in this case, periods of quartic hypersurfaces in $\mathbb{P}^3$). Our general methodology is obviously not tied to specific features of our hardware, but the computability scoring function we discuss here is inexorably biased by our available resources.

Experimentally we find that one critical subroutine of our larger algorithm $e \to \mathcal{P}_e$ tends to present a computational bottleneck. The global computation of $\mathcal{P}_e$ requires computing 21 ODEs and then numerically integrating them. In our 5-nomial dataset, the period transition matrix could only be computed for 2373 edges out of 5.97 million. An efficient way to identify these "needles in the haystack" is clearly needed. A quick way to identify a successful case is to consider only the first ODE out of 21, since in order for the computation to succeed all 21 ODEs need to be computed. In the same dataset, the first ODE could be computed for 112,380 edges, meaning that

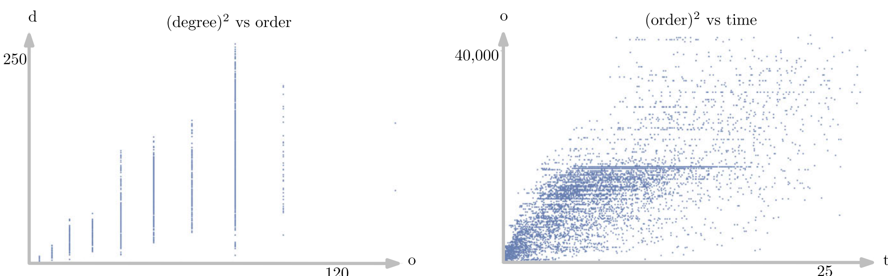


Figure 1. We experimented with including additional label information in training, such as the degree and order of the first ODE. However, we did not observe an advantage to using this additional data in training our neural networks. This may be attributed to the visible relation between time, degree, and order, shown here for the 5-nomial dataset. Notice the squared axes; this shows a very loosely quadratic relationship between time and order, and between order and degree

the conditional probability of success improves the baseline probability by a factor of approximately 53.

This observation leads to the following heuristic choice.

Definition 5.1. (*The subjective CSF*) Let ϕ' represent how long it takes to compute only the first of these ODEs on a fixed computer. That is, $\phi' \colon E \to \mathbb{R}_{\geq 0}$ is defined so that $\phi'(e)$ is the computation time for the first ODE required for $\mathcal{P}_e$. We define our *(subjective) computability scoring function (CSF)* as

$$\begin{aligned} \phi \colon E \to & \qquad [0,1] \\ e \mapsto & \min\{\phi'(e)/30, 1\}. \end{aligned}$$

Remark 5.2. Since $\phi'(e)$ can be arbitrarily large, we opt to terminate the computation of $\phi'(e)$ after 30 seconds. Our choice of a 30-second threshold is entirely heuristic, but necessary for the sake of practicality; even considering only the 4 or 5-nomial quartics with a 30-second threshold required roughly a CPU decade of computation time on our system to compute the CSFs (see Sect. 6.4) (Fig. 1).

We generate many training samples of ϕ, evaluated on edges of the graphs based on V_4 and V_5. We define a binary label $\beta(e) \in \{0, 1\}$, where $\beta(e) = 0$ if $\phi(e) < 1$ (*successful computation*, with respect to the chosen threshold of 30 seconds) and $\beta(e) = 1$ if $\phi(e) \geq 1$ (*failed computation*).

Definition 5.3. Define the *binary computability scoring function (BCSF)* as

$$\begin{aligned} \beta \colon E \to & \{0, 1\} \\ e \mapsto & 1 \text{ if } \phi(e) < 1, 0 \text{ otherwise}. \end{aligned}$$

We will refer to the resulting edge-time correspondences $\{(e, \beta(e))\}$ for V_k as *k-nomial data sets.* These pairs will be partitioned into training or

testing, in which case we may refer to e as an input data sample, and to $\beta(e)$ as that sample's label. Our architecture choice is in part guided by the structure of this data. In our learning task, it ends up being easier to learn the binary classification β rather than the nuanced function ϕ itself; in fact, in learning we obtain a proxy for ϕ given by the per-class probability distribution on E. Thus we replace our regression problem (approximate ϕ) with a binary classification problem (approximate β, and get a proxy for ϕ en route). We refer to these two functions interchangeably in the development of our strategy. We must be careful with how we represent an edge (f, g) (i.e, a pair of quartic surfaces); as the following section details, we are able to compute extra data associated to (f, g) that empirically relates to the complexity $\varphi(f, g)$.

5.3. Input Augmentation

We approximate the CSF via a statistical model that takes as input two features of an edge $e \in E$. We first represent each edge e by the concatenation of the coefficient vectors of the two polynomials that are the endpoints of that edge. It must be noted that in using this vector representation, we have discarded some information: the value of ϕ is linked inextricably to the fact that endpoints of e are polynomials, a characteristic of which the model is no longer aware. This guides the idea that our chosen learning method can better guess the value of $\phi(e)$ if it is provided more than just the coordinate representation of e.

One additional piece of domain information that we found useful was the *first-order* Gauss–Manin connection, see Sect. 2.5.2. We can efficiently compute the first-order Gauss–Manin connection at a few points of the pencil corresponding to e. In practice, the complexity of these matrices correlates with the computability score of e, as shown in Sects. 5.3.2 and 5.5.

We will use the following notation for these matrices. First, fix $t_1, \ldots, t_s \in \mathbb{C}$. For each e let $M_e = (M_{e,1}, \ldots, M_{e,s})$ be a sequence of such matrices, where $M_{e,i}$ is the first order Gauss–Manin connection evaluated at $t = t_i$ on the pencil corresponding to e. We will design our models to work with (e, M_e). In practice, we take $s = 1$ or $s = 2$ with $t_1 = 0$ and $t_2 = 1$ because we observed little benefit in increasing s further.

5.3.1. Studying the Complexity of the Input. The polynomials defining E have rational coefficients, and therefore the connection matrices M_e are also rational-valued. For example, when we are dealing with quartic surfaces defined over $\mathbb{Q}$, we have that (e, M_e) is an element of $\mathbb{Q}^n$, where $n = 2 \times 35 + s \times 21 \times 21$. In this section, we will consider various measures of complexity of the matrix M_e. Any correlation between "easy" statistics (defined in the next section) of M_e, and the algorithm-fragment runtime $\phi(e)$, is not obvious. This motivates the use of more expressive data-driven models.

5.3.2. Complexity of Cohomology Matrices. Because the standard implementation of neural networks work with floating point arithmetic, the subtleties of computing with a rational number are lost. As a large portion of the computation $e \mapsto \mathcal{P}_e$ is exact, the computation time is affected by the

TABLE 1. Three of the examined complexity measures

Measure	Definition
Sum	$\Psi_s(M) := \sum_{i,j,k} \psi(M_{ijk})$
Entropy	$\Psi_e(M) := -\sum_{ijk} \psi_{\text{entropy}}(M_{ijk})$
Length nonzero	$\Psi_l(M) := \text{len}(m \in \psi M : m \neq 0)$

"height" of the rational numbers involved. For this reason, we will modify the entries of M_e to better represent the complexity of its entries.

Let us define the following function on rational numbers:

$$\begin{aligned} \psi \colon \mathbb{Q} &\to \mathbb{R}_{\geq 0} \\ \frac{m_1}{m_2} &\mapsto \log(|m_1|) + \log(m_2) \end{aligned} \tag{5.2}$$

where $m_1, m_2 \in \mathbb{Z}$, $m_2 > 0$ and $\text{lcm}(m_1, m_2) = 1$. The value $\psi(m)$ of a rational number m is a more faithful representation of the complexity of computing with m then would be a floating point approximation of m. The following variation will also be used:

$$\psi_{\text{entropy}} \colon \frac{m_1}{m_2} \mapsto \log(|m_1|)^2 + \log(m_2)^2. \tag{5.3}$$

Various complexity statistics can be extracted from a complexity matrix M. Each statistic is a function $\Psi \colon \mathbb{Q}^{s\times 21\times 21} \to \mathbb{R}$. We list three options in Table 1.

Each column of Fig. 2 corresponds to an entry in Table 1. For each Ψ, we plot $\Psi(M_e)$ against the time it takes to compute the algorithm fragment defining ϕ, for those edges such that $\beta(e) = 1$. The edges for which $\beta(e) = 0$ are omitted from the top row of Fig. 2 because we had to terminate their computation prematurely, effectively assigning them all the same value of 30. The bottom row of Fig. 2 shows that successful edges do tend to have lower matrix statistics than failing edges. However, these distributions are not bimodal enough to make the statistics good classifiers in isolation.

On the other hand, we see no striking patterns between $\phi(e)$ and $\Psi(M_e)$ in the first row of Fig. 2. From this we conclude that the matrices M_e are useful in terms of classifying successes from failures (i.e. approximating β), but their statistics alone are not sufficient to regress (i.e. approximate ϕ) within the class of successes.

We can gain more insight by viewing each tensor M_e as a multi-channel image. We can visualize a matrix (M_{ij}) as a rectangular image with the ij-th entry colored a shade of blue: the darker the shading, the larger the value of $\psi(M_{ij})$. See Fig. 3, which has $M_{e,1}$ in the first row and $M_{e,2}$ in the second row for four e from the 4-monomial data set. We will see that this interpretation of the tensor as a multi-channel image lends itself naturally to learning with convolutional neural networks, as we wish to preserve the spatial relationships between matrix entries.

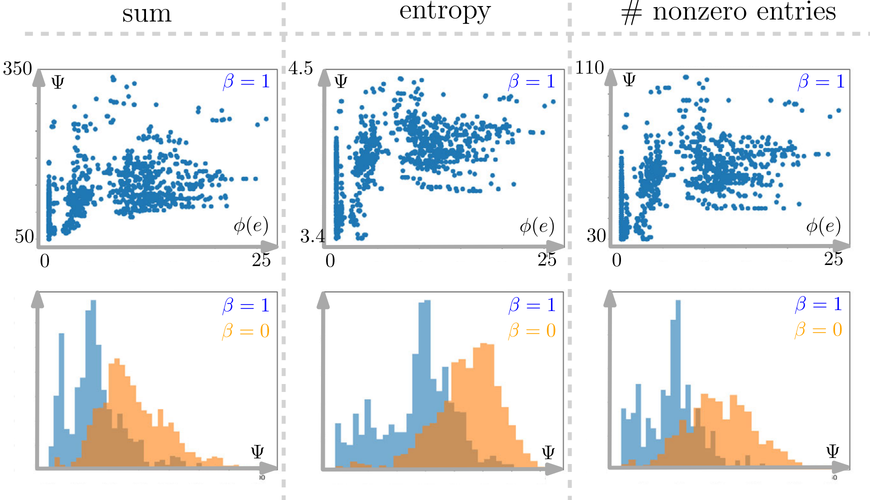


Figure 2. Complexity measure versus time. Dependence between the matrix statistics and computation time on the 4-nomial dataset. Top row: $\Psi(M_e)$ versus $\phi(e)$ on successful edges. Bottom row: Histograms of matrix statistics $\Psi(M_e)$, with the successful-edge distribution ($\beta(e) = 1$) denoted in blue, and the failing-edge distribution ($\beta(e) = 0$) denoted in orange (color figure online)

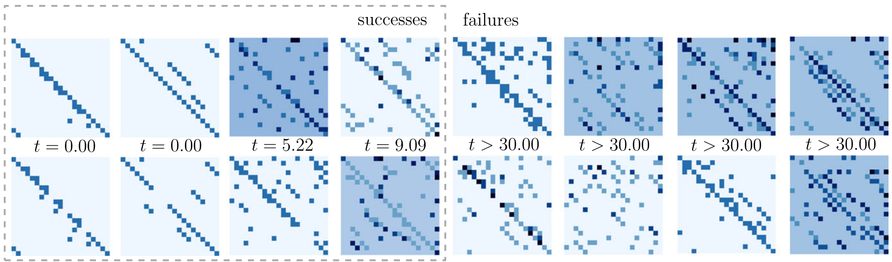


Figure 3. Inputs to the convolutional component of our ensemble. Top row: $\psi(M_{e,1})$, bottom row: $\psi(M_{e,2})$, with ψ in (5.2) evaluated entrywise. Darker shades indicate larger values of ψ

5.4. Dataset Preprocessing

5.4.1. Dimension Reduction: Principal Component Analysis. For the quartics we consider, the pair of polynomials in e for each $e \in E$ are sparse. As a result, a standard coordinate-space embedding represents e in an unnecessarily large (70-dimensional, for quartic surfaces) ambient space. This in turn, can complicate the learning problem: it can increase the number of parameters required to learn, and can introduce spurious local minima in

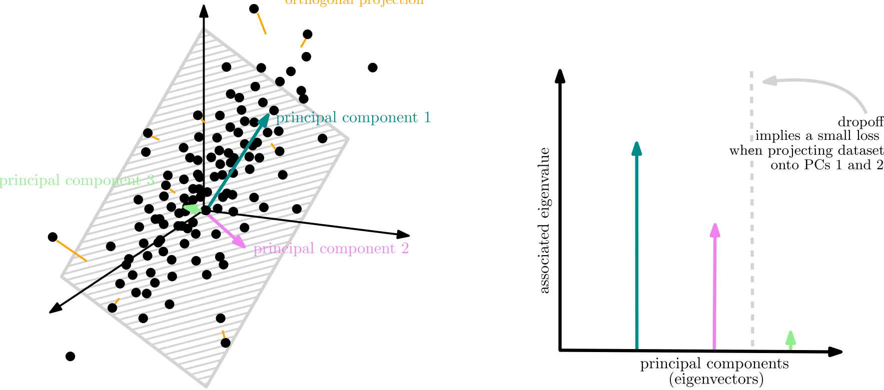


FIGURE 4. The eigenvalues (right) obtained during PCA describe the intrinsic dimensionality of the dataset (left). This can be used to effectively compress the dataset via orthogonal projection onto the most important principal components. The graphs in this figure are fictious and are for illustrative purposes only

the cost function. To reduce the dimension of the input space without losing information, we perform a *principal component analysis* (PCA) on a given E.

In short, PCA finds the "best fitting linear space" that almost contains all the points in E. The process also minimizes the dimension of this best fitting linear space. More precisely, PCA is an orthogonal linear transformation that compresses the data stored in e, by projecting it onto a subspace with minimal loss of dataset variance. For more information about PCA and its derivation, see [30].

The "principle components" are the eigenvectors of a covariance matrix of the dataset. Each associated eigenvalue describes how much the dataset aligns with the direction of the corresponding eigenvector. Thus, the relative "importance" of each direction in data compression can be found by reading off these eigenvalues.

A pictorial example of this concept is given in Fig. 4, where we can associate a significant dropoff in covariance-matrix eigenvalues to an ability to losslessly compress the dataset. In this example, the point cloud is nearly planar, and this is reflected in the sudden dropoff in eigenvalues on the right.

Figure 5 shows that from the first 23 principal components, one can almost entirely recover the coordinates of edges in the complete graph over V_4. This value of 23 also appears to apply to V_5. We thus compress each e to a 23 dimensional vector, decreased from 70 dimensions. In doing so, we make no claim about this new 23-dimensional space being interpretable as the same kind of polynomial space as before; it is an abstract albeit convenient coordinate space.

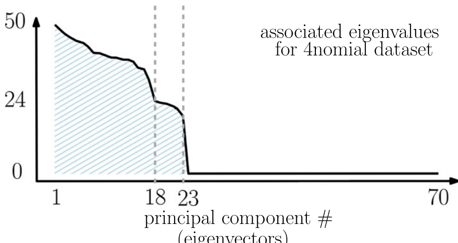


FIGURE 5. The distribution of the eigenvalues coming from the PCA applied to the complete 4-nomial graph dataset

5.4.2. Balancing the Dataset. A mark of how difficult the integration step can be is that even the algorithm fragment of Sect. 5.2 fails to terminate in the vast majority of pairs we considered. As a result, a training set $\mathcal{T}$ as in (5.1) will consist almost entirely of failed edges of the form $(e, 0)$. This incentivizes the neural network to produce the constant 0 function as an approximation of ϕ. As a remedy, we employ a standard method to disincentivize the neural network from converging to a constant. This method is done by over-sampling the under-represented class (here, edges e with $\beta(e) = 1$) so that the training set consists of an equal portion of both classes.

5.5. Comparison of Learning Models

As with many learning challenges, a crucial decision is whether to use neural networks at all. Classical statistical methods are simpler to implement, but neural networks can be preferable for vision-related tasks. Section 5.5.1 demonstrates that in the context of this problem, the neural networks we have tried consistently outperform the classical methods. The discussion in this section will center around the 4-monomial dataset, i.e. the complete graph on V_4.

We experimented with several regression techniques as candidates for learning the quite irregular CSF ϕ, and found that we could get a better estimate for ϕ through a simple *binary classification*, i.e. approximation of the BCSF β. The methods we consider output functions $f : E \to [0, 1]$ that map to probabilities of edges being "computable". We turn the function f into a binary classifier by choosing a cut-off value τ such that $f(e) \leq \tau$ should be correlated with $\beta(e) = 0$.

We will compare the performance of various statistical models via their *receiver operating characteristic (ROC) curves* [11] on the 4-nomial dataset. We direct the reader to [1] for a comprehensive text that defines and discusses the classical statistical models we used in this section. An ROC curve is a standard tool to compare the performance of binary classifiers on the same dataset. Each curve corresponds to a single binary classifier, and can be traced by varying the threshold τ from 0 to 1. In this curve, performance is measured by comparing the classifier's true positive rate and false positive rate. The closer a curve is to the top left vertex, i.e. the point $(0, 1)$, the better the corresponding method performs. The dotted line is the idealized

FIGURE 6. Receiver operating characteristic curves of various choices of "computability scoring functions" (CSF), on 4-nomials. Given a CSF, a choice of threshold $\tau \in [0, 1]$ gives a *binary* CSF which in turn yields a single True/False Positive rate. Each curve is formed by varying this threshold value τ. The top performers, i.e. those reaching closest to the point $(0, 1)$, are deep learning methods

curve for the method of random guessing. The ROC curves in Fig. 6 show the ranking of ten different methods. The best-performing method, *deep ensemble*, is a composition of the next best two neural network strategies, namely a multilayer perceptron and a convolutional neural network. The remaining seven are classical methods, all of which underperform in this task.

5.5.1. Classical Statistical Methods. We used seven standard binary classifiers to approximate ϕ using the 4-nomial training set $\mathcal{T}$. Since they were clearly outperformed by the neural networks, we will not discuss them in depth. The methods we used are: logistic regression with an L_2 penalty; a regularized support vector classifier with a linear kernel and regularization parameter 1; a regularized support vector classifier with degree-2 radial basis function kernel and regularization parameter 1; a random forest with 10 trees, per-tree maximum depth 5; an AdaBoost classifier with at most 50 estimators; linear discriminant analysis; and Gaussian Naive Bayes. Figure 6 suggests that the deep classifiers will outperform the classical methods. This may simply be caused by our deep learning methods having more internal parameters and therefore providing better approximations.

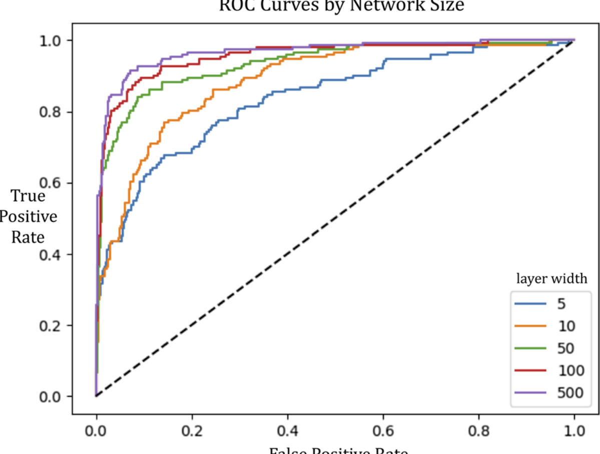


Figure 7. Architecture exploration for the MLP on the 4-nomial data. The key associated to this set of ROC curves denotes the widths for the first three layers of the MLP. This data is helpful in selecting a network size

5.5.2. Deep Neural Networks. We will introduce two neural models and then combine their results into an ensemble method. The first model is a multi-layer perceptron (MLP), which follows the standard formulation in Sect. 4. In dealing with the 4-nomial dataset, we decided on an architecture with five hidden layers, each of width 100. As input, it takes only the edges $e \in E$ after compression via principal component analysis as in Sect. 5.4.1. Its output is a single value in $\mathbb{R}$. To restrict the codomain to the interval $[0, 1]$ we apply the inverse logit function. After training, the neural network gives an approximation $\phi_{\mathrm{MLP}}\colon E \to [0, 1]$ of the computability score. Figure 7 shows the consequence of changing the dimension of the first three hidden layers. Other parameters have also been chosen by considering such figures to optimize predictive power against performance.

Our second neural network is a two-channel convolutional neural network, a variation of the standard neural network explained in Sect. 4 to better detect patterns in visual data. This neural network will be trained using the 3-tensors M_e encoding the first order Gauss–Manin connections. We will however apply the complexity function ψ from Sect. 5.3.2 to each entry of M_e before giving it as input. The output is adjusted as with the first neural network (MLP) above so that, after training, we obtain an approximation $\phi_{\mathrm{CNN}}\colon E \to [0, 1]$. More details about both network architectures are available in the supplementary code.[3]

We improve on the approximations of the two neural networks by defining the function

$$\phi_{\text{ensemble}} := \phi_{MLP} \cdot \phi_{CNN}. \tag{5.4}$$

[3]See https://github.com/a-kulkarn/period_graph.

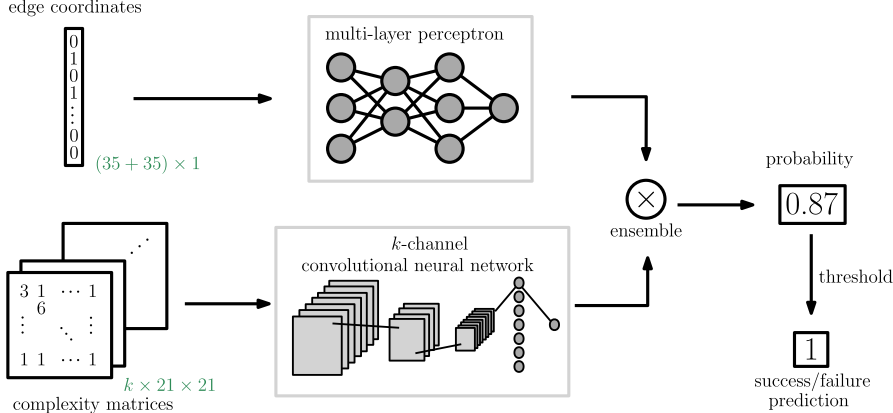


FIGURE 8. Schematic of the deep ensemble binary classifier

This is what we call the "deep ensemble" method (or just ensemble method), as illustrated in Fig. 8. Being the product of two functions, which are essentially probability functions, the function ϕ_{ensemble} is more cautious in returning a value close to 1. We chose this approach because attempting the computation $e \mapsto \mathcal{P}_e$ for edges that do not terminate can be very costly. We prefer a computability score that has a low false positive rate.

5.6. Implementation

We now describe our experimental setup and the performance of our neural learners. Recall that V_n is the set of smooth quartics that are the sum of n monomials.

5.6.1. Four-Monomial Quartics. Consider the edges E of the complete graph with vertices V_4. Note $\#V_4 = 108$ and $\#E = \binom{108}{2} = 5778$. These numbers are so small that we could run our test computation (as in Sect. 5.2) on all edges in E. This gives us complete information and allows us to evaluate the performance of our neural network.

For $\alpha \in (0,1) \subset \mathbb{R}$ take a random subset $E' \subset E$ for which $\#E' \sim \alpha \cdot \#E$. We train the ensemble neural network defined in (5.4) on E' and test it on $E'' := E \setminus E'$. We wish to know whether the neural network correctly predicts whether the first ODE (out of 21) associated to an edge $e'' \in E''$ can be successfully computed within our prescribed threshold of 30 seconds. For various values of α we list in Table 2 the percentage of *true negatives (TN), false positives (FP), false negatives (FN) and true negatives (TN)*—the "positive" label means that the ODE computation for the edge can be completed within 30 seconds. The ensemble classifier learns to predict the answers with good accuracy even with training on a small fraction of the available dataset.

As discussed for input dimension reduction, having too many network parameters (i.e. weights and biases) can lead to unwieldy computations and local minima. We see this behavior when we work with networks that are

Table 2. The performance of the ensemble neural network on V_4 with varying proportion α of edges used for training

α	TN (%)	FP (%)	FN (%)	TP (%)	(TP+TN)/(FP+FN)
0.3	74.87	4.43	7.42	13.27	7.43
0.5	68.38	3.87	6.97	20.78	8.22
0.7	64.54	2.59	6.76	26.11	9.70

Table 3. The performance of the ensemble neural network on V_5

α	TN (%)	FP (%)	FN (%)	TP (%)	(TP+TN)/(FP+FN)
0.3	83.06	1.80	0.69	14.44	39.05
0.5	84.48	1.63	0.49	13.40	46.19
0.7	83.70	1.52	0.46	14.32	49.48

too large and we find small networks not to be expressive enough. To find the appropriate size, we compare the performance of networks with different sizes, and choose the smallest such model with good accuracy.

Taking $\alpha = 0.9$, on Fig. 7 we plotted the ROC curves of the multilayer perceptron component of the ensemble for different widths of its first three layers. We decided on a width of 500 based on this figure.

5.6.2. Extrapolating from Four-Monomial to Five-Monomial Quartics. The performance of the ensemble method trained solely on V_4 did not perform well on V_5, an observation that we did not find particularly surprising. One plausible reason is that the first order Gauss–Manin connections on V_4 are all quite simple, whereas on V_5 there is a broader range of complexity displayed by these matrices. In other words, the structure in set V_4 does not necessarily extrapolate well to V_5.

To address this, we retrained the ensemble method on 5-monomials. Consider the edges E of the complete graph with vertices V_5. This is a much larger set, since $3348 = \#V_5 \gg \#V_4 = 108$, so $\#E = 5,602,878$. With the ratio α defined as in Sect. 5.6.1, we display the performance for various α on Table 3.

6. Application

In this section we will give an application of our software package and analyze the performance improvement of using neural networks.

6.1. Five Monomials

One application of our software is to the set V_5 of smooth polynomials, each of which are the sum of five coefficient-1 monomial terms; see Sect. 3.1.2 for

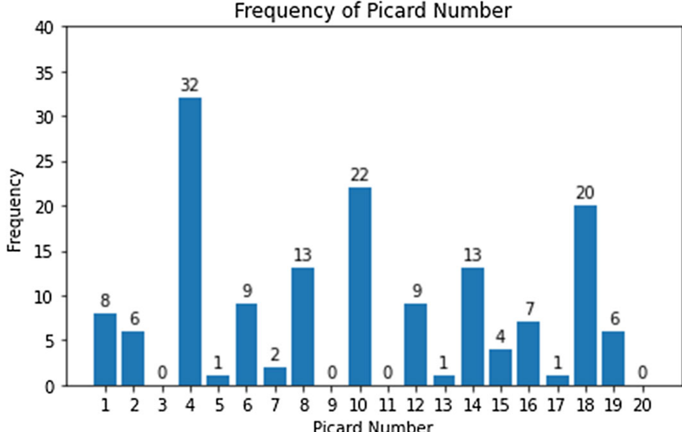


Figure 9. Picard number frequency for 5-nomial quartics. This histogram does not include any 4-nomial data

this notation. Our goal in this section is to compute the Picard numbers of these 5-nomial quartics and to find unexpected isomorphisms in this set.

6.2. List of Results

We first note that the results of this section depend on finding integral linear relations between periods that are only known approximately. Therefore, the results below may contain errors due to insufficient precision; we have attempted to mitigate this risk by working with 300 digits of precision, when we could attain it.

Of 3348 smooth 5-nomial quartics, there are only 161 S_4-symmetry classes, and of this 161 we were able to reach 154. Their Picard numbers and frequencies are listed in the following table.

In contrast to the 184,000 quartics in the database given in [22], it seems we are targeting a small set of quartics. Indeed, there are only 161 smooth quartics that are the sum of five monomials upto permuting the variables. However, the database in [21] was built by random exploration to find quartics that are easy to compute. Moreover, that set contains only 127 isomorphism classes of quartics! Here, we found 139 distinct isomorphism classes amongst the 154 quartics we could reach. The main difference is that our target set of quartics V_5 is fixed, and includes quartics that are difficult to reach. We are thus forced to use an order of magnitude more computation time to build our graph (a CPU year versus decade). In fact, some of these quartics are so difficult to reach that our "optimal" tree T spanning 154 S_4-classes has a diameter of length 21.

Remark 6.1. The Picard number for every 5-nomial quartic in this database was verified using a complementary method that uses the crystalline cohomology of finite reductions [7]. This method produces guaranteed upper bounds. For each of our polynomials, we ran through as many prime reductions as

Table 4. Picard numbers for the 7 missing quartics

Polynomial	Picard number
$x^3y + y^3z + y^3w + z^3w + xw^3$	≤ 2
$xy^3 + z^4 + x^3w + y^2zw + xw^3$	≤ 2
$x^4 + y^3z + xz^3 + x^3w + yw^3$	≤ 2
$y^3z + xyz^2 + xz^3 + x^3w + yw^3$	≤ 3
$x^3y + y^3z + z^3w + z^2w^2 + xw^3$	≤ 3
$x^2y^2 + x^3z + yz^3 + y^3w + xw^3$	≤ 18
$xy^3 + x^3z + xyzw + z^3w + yw^3$	≤ 19

was necessary to have the minimum of the upper bounds attained thrice. In each case, this minimum agreed with the Picard numbers we computed.

Remark 6.2. The method using crystalline cohomology took about 10 minutes on average per polynomial. This compares favorably to computing period transition matrices, which can take anywhere between seconds and hours (including finding a good path of deformation). On the other hand, using the periods we can compute the entire Picard lattice (not just its rank), compute endomorphism fields, and isomorphism classes.

Remark 6.3. Using the Picard lattice, we can use [22, §3] to find smooth rational curves in the surface. For instance, all but seven of the 154 quartics had Picard groups that could be generated over $\mathbb{Q}$ by classes of smooth rational curves of degree ≤ 3 and the hyperplane class. For the remaining seven, two of them need smooth rational curves of degree ≤ 4, three need degree ≤ 5, and two need ≤ 6 (in addition to the hyperplane class).

6.2.1. The Missing Vertices. While our search method succeeded for the vast majority of quartics, we were unable to reach seven of the 5-nomial quartics in V_5, up to isomorphism. These seven quartics are listed in Table 4. Moreover, we used `crystalline_obstruction` [7] as explained in Remark 6.1 to give upper bounds for their Picard numbers.

We also tried to brute force every edge from $V_4 \cup V_5$ to establish a connection to the three quartics in this table with Picard number 2. In each connection attempt, we allowed the computation for the first ODE to proceed for 3 hours before terminating; even with this generous threshold, we only succeeded in connecting these vertices to one another but not to the main component. Naturally we could not try every edge with the three hour time limit. Therefore, it is conceivable but very unlikely that good connections exist.

This prompts us to ask: *Which feature of these quartics is responsible for making them inaccessible?*

6.2.2. Isomorphism Classes. Using the Torelli theorem for K3 surfaces [26] we can check if the K3 surfaces in our list admit non-trivial isomorphisms. Here, we work only with 154 (of 161) representatives of the S_4-symmetry classes

Table 5. The 11 non-trivial isomorphism classes

$y^3z + yz^3 + x^3w + xw^3 + w^4$	$y^4 + z^4 + x^3w + xw^3 + w^4$
$y^4 + y^2z^2 + z^4 + x^3w + w^4$	$y^3z + y^2z^2 + yz^3 + x^3w + w^4$
$y^4 + z^4 + x^3w + yzw^2 + w^4$	$y^3z + yz^3 + x^3w + yzw^2 + w^4$
$y^4 + z^4 + x^3w + xz^2w + xw^3$	$x^4 + y^4 + z^4 + yzw^2 + w^4$
$y^4 + z^4 + x^3w + xyzw + w^4$	$y^3z + yz^3 + x^3w + xyzw + w^4$
$x^4 + y^4 + z^4 + zw^3 + w^4$	$y^4 + yz^3 + z^4 + x^3w + xw^3$
$y^4 + x^2yz + z^4 + x^3w + w^4$	$y^4 + yz^3 + x^3w + xz^2w + xw^3$
$y^3z + y^2z^2 + z^4 + x^3w + xw^3$	$y^4 + z^4 + x^3w + x^2w^2 + w^4$
$y^3z + yz^3 + x^3w + yzw^2 + xw^3$	$y^4 + z^4 + x^3w + yzw^2 + xw^3$
$y^3z + y^2z^2 + yz^3 + x^3w + xw^3$	$y^4 + y^2z^2 + z^4 + x^3w + xw^3$
$y^4 + z^4 + x^3w + x^2w^2 + xw^3$	$x^4 + y^4 + z^4 + z^2w^2 + w^4$
$x^4 + y^4 + z^4 + xyzw + w^4$	$y^4 + z^4 + x^3w + xyzw + xw^3$
$y^3z + yz^3 + x^3w + xyzw + xw^3$	$y^4 + xz^3 + x^3w + xyzw + zw^3$

Table 6. Frequency of endomorphism fields

Frequency	Number field
60	$t - 1$
35	$t^2 + 1$
43	$t^2 + t + 1$
8	$t^4 - t^2 + 1$
7	$t^6 + t^3 + 1$
1	$t^{12} - t^6 + 1$

for which we could compute the periods. We compared their period vectors modulo an integral change of basis for homology. The method of computation is described in Sect. 6.3.2. We found 9 isomorphism classes of size 2 and 2 isomorphism classes of size 4, all other isomorphism classes appear to be of size 1. This gives 139 isomorphism classes in total. We display the non-trivial isomorphism classes in Table 5.

6.2.3. Endomorphism Fields. We also computed the endomorphisms of the transcendental lattice of a K3 from its periods. We use the argument in Sect. 6.3.2 with $X_1 = X_2$ in order to compute these. The endomorphism ring E of the transcendental lattice of a K3 is always a field, either of real or complex multiplication [34]. Except when $E = \mathbb{Q}$, we did not observe any real multiplication surfaces. These are notoriously hard to find [10]. In Table 6 we list the polynomials $f(t)$ for which $E \simeq \mathbb{Q}[t]/f(t)$ and the number of times this endomorphism was realized among our 154 S_4-symmetry classes in V_5.

6.3. Methodology

Once the period matrix of a quartic is approximated, we follow [22] to compute the Picard numbers. In order to facilitate the computation of the periods, there are two tricks we used besides the general strategy outlined in Sect. 3.1.3.

6.3.1. Additional Simplifications. The symmetric group S_4 acts on the 5-nomials by permuting the four variables. This is a linear action of the projective space and we can use Sect. 2.5.1 to compute period translation matrices at essentially no cost. This connects the elements in each S_4-equivalence class.

In order to translate the periods of one polynomial p to another q, we need the period matrix of p. However, if p is particularly resistant to our computations then we can compute only the first row of the period matrix of p—reducing the work load by a factor of 21. With this first row we are still able to compute the Picard number and isomorphism class of p. However, p becomes a dead-end; we can no longer use p to compute the periods of another polynomial q.

6.3.2. Computing Isomorphism Classes. The isomorphism class of a K3 surface depends only on its periods [26]. In particular, that of the first row of its period matrix.

Suppose $w_1, w_2 \in \mathbb{C}^{22} \simeq \mathrm{H}^2(X, \mathbb{C})$ are periods of two K3s X_1 and X_2. To detect if X_1 and X_2 are isomorphic, we need to determine if there exists a constant $c \in \mathbb{C}^*$ and an isometry $N \in \mathbb{Z}^{22\times 22}$, $N\colon \mathrm{H}_2(X_1, \mathbb{Z}) \xrightarrow{\sim} \mathrm{H}_2(X_2, \mathbb{Z})$, such that

$$w_1 \cdot N = cw_2. \tag{6.1}$$

Using approximations of w_1 and w_2, this can be translated into a problem of finding short lattice vectors as we describe below.

The integral relations annihilating w_1 and w_2 cause a difficulty here. So we first compute the Picard groups $\mathrm{Pic}(X_i) \simeq w_i^\perp \subset \mathbb{Z}^{22}$. If the rank of the Picard groups are distinct then X_1 and X_2 are not isomorphic.

If $\rho := \mathrm{rk}\,\mathrm{Pic}(X_1) = \mathrm{rk}\,\mathrm{Pic}(X_2)$, construct $T(X_i) = \mathrm{Pic}(X_i)^\perp \subset \mathbb{Z}^{22}$. We can view w_i as an element in $T(X_i) \otimes \mathbb{C} \simeq \mathbb{C}^{22-\rho}$. Let $v_i \in \mathbb{C}^{22-\rho}$ be the new vector corresponding to w_i. The surfaces X_1 and X_2 are isomorphic if and only if there exists $c \in \mathbb{C}^*$ and $N' \in \mathbb{Z}^{22-\rho\times 22-\rho}$ that satisfy

$$v_1 \cdot N' = cv_2. \tag{6.2}$$

If such an N' exists, then $\langle v_2, v_1 \cdot N'\rangle = \langle v_2, v_2\rangle c$, where $\langle\cdot,\cdot\rangle$ denotes the intersection product on $T(X_2)$. Then,

$$\langle v_2, v_1 \cdot N'\rangle v_2 = \langle v_2, v_2\rangle v_1 \cdot N', \tag{6.3}$$

which eliminates the unknown c. As in [22, §2.3] we use LLL [24] to find possible integral solutions for (6.3) in N'. The function `isomorphisms_of_k3s` in `PeriodSuite` implements this procedure.

6.4. Performance on Applications

Our objective in this work was to use a neural ensemble method to improve the computation time for periods. In Sect. 5.6, we analyzed the predictive power of the ensemble method on a dataset that we were able to label via substantial computational effort. In this section, we apply the ensemble classifier to a dataset more reflective of the intended use-case. Specifically, we explore the classifier's ability to predict which edges between 5-nomial and 6-nomial quartics are traversable. The number of edges in the complete bipartite graph with vertex sets (V_5, V_6) is far too large for us to generate a traversable/intraversable label for every edge. Our results in this section indicate that *transfer learning* is effective in detecting traversable edges between 5-nomial and 6-nomial quartics.

We demonstrate the effect of using our ensemble method on two small examples (Figs. 10, 11). The problem here was faced on a larger scale, and faced repeatedly, as we sought to complete the calculations for Fig. 9. Given a 5-nomial $f \in V_5$, we look for 4-or 6-nomials $g \in V_4 \cup V_6$ such that the period transition matrix for the edge (f, g) is easy to compute. We used such connections to zig-zag from V_5 to V_4 or V_6 and back to V_5 in order to establish new connections between 5-nomials.

For this example, we chose a random subset $S \subset V_5$ with 100 elements so that the elements of S are pairwise distinct under the action of S_4, i.e., $\#(S/S_4) = 100$. For each $f \in S$ we consider the edge set

$$E_f = \{(f, g) \in V_5 \times V_6 | f - g \text{ is a monomial}\}. \tag{6.4}$$

The average size of E_f for $f \in S$ is 29. Such an edge set is illustrated in Fig. 12.

We compare two methods of exploring the edges E_f, one aided by our neural model and one unaided. We used the neural model that was trained on V_5 but not on V_6 so that there is no extra training time. In particular, *the neural model is faced with a data set for which it has not been trained*; nevertheless it performs well.

For the unaided strategy, we picked 10 elements from each E_f randomly and tried to compute these edges. For the aided strategy, we sorted E_f using our neural network and picked the top 10. The unaided strategy had a 54.4% failure rate as opposed to 33.9% for the aided strategy. Consider the table below that records the frequency of elements in S that had n successful edges for $n \in \{0, \ldots, 10\}$. The first column of this table shows that, in both cases, 21 vertices had 0 successful edges.

This demonstrates the fact that some vertices are intrinsically difficult to move away from; this reminds us even the best statistical model will not succeed if every edge is impossible. On the other hand, we see that the aided method establishes far more connections to V_6. For instance, 47 vertices in S had all 10 of their chosen edges successful with the aided method as opposed to 13 with the unaided method. In practice, this computation would then be repeated for each successful connection, which means that the advantage grows exponentially.

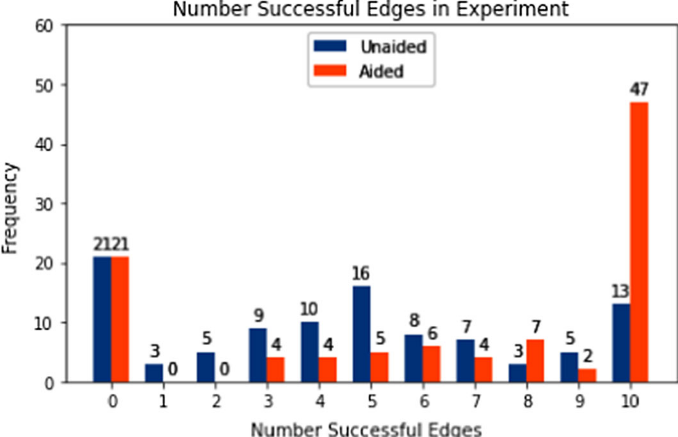


FIGURE 10. We demonstrate the efficacy of our proposed algorithm with the following experiment: A trial consists of the following steps: (1) Choose a random central vertex given by a sparse polynomial f. (2) Construct the set of edges E_f as in (6.4). (3) Select 10 edges (orange: using our neural model, blue: selected randomly) and count how many are successful. Figure 12 is an illustration of a single trial. The histogram above depicts the results of 100 trials using each strategy. The aided strategy is a substantial improvement over the unaided strategy, achieving a maximum score of 10 far more often than random

The second experiment is analogous to the first one. Except the random set of vertices S is chosen from V_6, the set of edges is constructed similarly to (6.4); for each $f \in S$ we consider

$$E_f = \{(f, g) \in V_6 \times V_6 | \#\text{terms}(f - g) = 2\}. \tag{6.5}$$

The result is given in Fig. 11. This result is more striking since the neural network was not trained on 6-nomials and yet it clearly outperforms the random method. Once again, a large portion of the edges are inaccessible to our methods.

Our computations for the Picard ranks of V_5 took a CPU decade. The approach presented in this paper allowed us to repeatedly pare down hundreds of thousands of possible edges to a manageable, but likely to succeed, subset. The mini-computation in this section demonstrates the benefit of including a neural ensemble model in the algorithmic pipeline en route to computing period matrices of smooth quartic hypersurfaces.

6.5. Costs of Collecting the Training Data and Training the Network

The training data for our neural network comes essentially for free, in the sense that, even an "unaided" period computation method that randomly searches for good deformations will produce the training data as a byproduct.

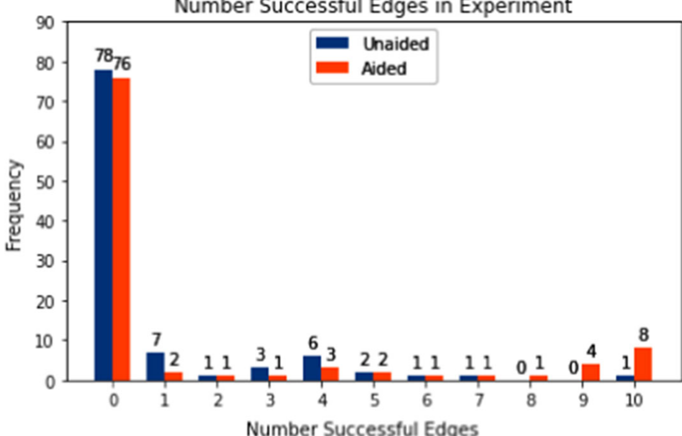


FIGURE 11. We conduct a trial similar to the experiment in Fig. 10. This time we pick $S \subset V_6$ as a random subset of 100 6-nomial quartics and construct edges in $V_6 \times V_6$ as in (6.4). The neural network was not trained on 6-nomials. The histogram depicts the results of 100 trials using each strategy. Again, the aided strategy is a substantial improvement over the unaided strategy. Once again, many edges appear to be inaccessible

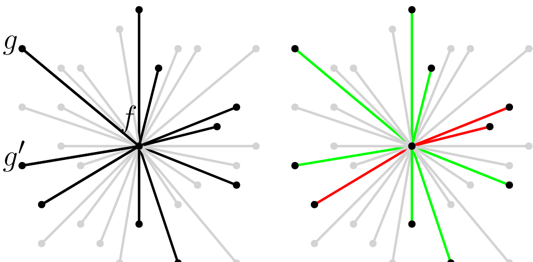


FIGURE 12. Left: Illustration of a set of edges E_f according to our sampling method. Right: A randomly sampled subset of 10 edges, with 7 successful edges and 3 failed edges

For our 4-and-5-nomial dataset set, we performed 10 CPU years of computations. Our neural network was trained on the by-product of this computation. In comparison, the time to train the neural model took 21 CPU days. Clearly, the cost of training is negligible in comparison to the primary computations. For this reason, and with the results of Sect. 6.4 in mind, the neural network has a net positive impact on period computations.

We now clarify the strategy of training a neural network alongside a random search to achieve this net positive impact. Consider a graph $G = (W, E)$ in which we are trying to compute the periods of some (or all) of the vertices. Without prior training data, we need to begin exploring G by trying

to compute period transition matrices for randomly selected $e \in E$. The cohomology matrices associated to an edge are a by-product of the period transition computation, and the label for the edge is known based on the result of the computation. Over time, the exploration process accumulates a labelled dataset that can be used to train a neural model. After a comparatively short time for training, the neural model can be deployed to accelerate the search for traversable edges in G. This process can be repeated as training data grows larger.

Acknowledgements

We would like to thank the referees for a careful reading of the manuscript and for their valuable comments. We are grateful to the following institutions for allowing us to use their computational resources: Dartmouth College, Harvard University, Leibniz University Hannover, and Max Planck Institute for Mathematics in the Sciences (MPI MiS) Leipzig. This project began while all three authors were at MPI MiS. We thank MPI MiS for providing a stimulating environment. Also, we thank Pierre Lairez for his helpful comments.

Funding All three authors were partially funded by Max Planck Institute for Mathematics in the Sciences (MPI MiS) Leipzig. In addition to that: K. Heal was funded by Harvard University and C. S. Draper Laboratory. A. Kulkarni was funded by Dartmouth College and the Simons Collaboration on Arithmetic Geometry, Number Theory, and Computation (Simons Foundation grant 550033). E.C. Sertöz was funded by Leibniz University Hannover. All of the educational institutions provided computational resources.

Declarations

Conflict of interest There is no conflict of interest.

Data availability The periods of all the few monomial quartics that we have been able to compute are available on our Dropbox repository. We also included our Picard lattice computations in that repository.

Code availability The software package developed for this project is available at github: period_graph. This software depends on the custom package PeriodSuite. These packages are written to work in an environment running SageMath [9] and Magma [3].

Open Access. This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by

statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

[1] Aggarwal, C.C.: Data classification. In: Data Mining, pp. 285–344. Springer (2015)

[2] Arapura, D.: Algebraic Geometry over the Complex Numbers. Springer, Berlin (2012)

[3] Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system. I. The user language. J. Symb. Comput. **24**(3–4), 235–265 (1997). Computational algebra and number theory (London, 1993)

[4] Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest paths algorithms: theory and experimental evaluation. Math. Program. **73**(2), 129–174 (1996)

[5] Claesen, M., De Moor, B.: Hyperparameter search in machine learning. Proc. of the 11th Metaheuristics International Conference (2015)

[6] Costa, E., Harvey, D., Kedlaya, K.S.: Zeta functions of nondegenerate hypersurfaces in toric varieties via controlled reduction in p-adic cohomology. In: Proceedings of the Thirteenth Algorithmic Number Theory Symposium, pp. 221–238. Mathematical Sciences Publishers (2019)

[7] Costa, E., Sertöz, E.C.: Effective obstruction to lifting Tate classes from positive characteristic. In: Arithmetic Geometry, Number Theory, and Computation. Simons Symposia. Springer (2022)

[8] Cybenko, G.: Approximation by superpositions of a sigmoidal function. Math. Control Signals Syst. **2**(4), 303–314 (1989)

[9] Developers, T.S.: SageMath, the Sage Mathematics Software System (Version 9.3) (2021). http://www.sagemath.org

[10] Elsenhans, A.S., Jahnel, J.: Examples of K3 surfaces with real multiplication. LMS J. Comput. Math. **17**(A), 14–35 (2014). https://doi.org/10.1112/S1461157014000199

[11] Fawcett, T.: An introduction to ROC analysis. Pattern Recognit. Lett. **27**(8), 861–874 (2006)

[12] Girosi, F., Poggio, T.: Networks and the best approximation property. Biol. Cybern. **63**(3), 169–176 (1990)

[13] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). http://www.deeplearningbook.org

[14] Griffiths, P.A.: On the periods of certain rational integrals. I. Ann. Math. (2) **90**, 460–495 (1969)

[15] Griffiths, P.A.: On the periods of certain rational integrals. II. Ann. Math. (2) **90**, 496–541 (1969)

[16] He, Y.H., Kim, M.: Learning algebraic structures: preliminary investigations (2019). arXiv:1905.02263

[17] He, Y.H., Lee, S.J.: Distinguishing elliptic fibrations with AI. Phys. Lett. B **798**, 134889 (2019). https://doi.org/10.1016/j.physletb.2019.134889

[18] Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Netw. **2**(5), 359–366 (1989)

[19] Huang, Z., England, M., Wilson, D.J., Bridge, J., Davenport, J.H., Paulson, L.C.: Using machine learning to improve cylindrical algebraic decomposition. Math. Comput. Sci. **13**(4), 461–488 (2019)

[20] Katz, N.M., Oda, T.: On the differentiation of De Rham cohomology classes with respect to parameters. J. Math. Kyoto Univ. **8**, 199–213 (1968)

[21] Lairez, P., Sertöz, E.: A database of quartic surfaces. http://pierre.lairez.fr/quarticdb/ (Accessed 21 Jan 2022) (2018)

[22] Lairez, P., Sertöz, E.C.: A numerical transcendental method in algebraic geometry: computation of Picard groups and related invariants. SIAM J. Appl. Algebra Geom. **3**(4), 559–584 (2019). https://doi.org/10.1137/18M122861X

[23] Lairez, P., Sertöz, E.C.: Separation of periods of quartic surfaces (2020). arXiv:2011.12316

[24] Lenstra, A.K., Lenstra, H.W., Jr., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. **261**(4), 515–534 (1982)

[25] Leshno, M., Lin, V.Y., Pinkus, A., Schocken, S.: Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Netw. **6**(6), 861–867 (1993)

[26] Looijenga, E., Peters, C.: Torelli theorems for Kähler $K3$ surfaces. Compos. Math. **42**(2), 145–186 (1980)

[27] Movasati, H., Sertöz, E.C.: On reconstructing subvarieties from their periods. Rendiconti del Circolo Matematico di Palermo Series 2 (2020). https://doi.org/10.1007/s12215-020-00562-x

[28] Russell, S., Norvig, P.: Artificial intelligence: a modern approach. Prentice Hall series in artificial intelligence. Pearson Education (2016)

[29] Sertöz, E.C.: Computing periods of hypersurfaces. Math. Comput. **88**(320), 2987–3022 (2019)

[30] Shlens, J.: A tutorial on principal component analysis. Int. J. Adv. Res. Comput. Sci. Manag. Stud. (2005)

[31] Sirignano, J., Spiliopoulos, K.: DGM: a deep learning algorithm for solving partial differential equations. J. Comput. Phys. **375**, 1339–1364 (2018). https://doi.org/10.1016/j.jcp.2018.08.029

[32] Voisin, C.: Hodge theory and complex algebraic geometry. I, Cambridge Studies in Advanced Mathematics, vol. 76, English edn. Cambridge University Press, Cambridge (2007). Translated from the French by Leila Schneps

[33] Voisin, C.: Hodge theory and complex algebraic geometry. II, Cambridge Studies in Advanced Mathematics, vol. 77, English edn. Cambridge University Press, Cambridge (2007). Translated from the French by Leila Schneps

[34] Zarhin, Y.G.: Hodge groups of $K3$ surfaces. J. Reine Angew. Math. **341**, 193–220 (1983)

Kathryn Heal
School of Engineering and Applied Sciences
Harvard University
29 Oxford Street
Cambridge MA02138
USA
e-mail: kathematical@gmail.com

Avinash Kulkarni
Department of Mathematics
Dartmouth College
Kemeny Hall, 27 N Main St
Hanover NH03755
USA
e-mail: avinash.a.kulkarni@dartmouth.edu

Emre Can Sertöz
Institut für Algebraische Geometrie
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover
Germany
e-mail: emre@sertoz.com

Received: August 18, 2021.
Accepted: January 25, 2022.