Differentiable Spline Approximations

Minsu Cho[1†] **Aditya Balu**[2†] **Ameya Joshi**[1] **Anjana Deva Prasad**[2]

Biswajit Khara[2] **Soumik Sarkar**[2] **Baskar Ganapathysubramanian**[2]

Adarsh Krishnamurthy[2] **Chinmay Hegde**[1]

New York University[1], **Iowa State University**[2]
{mc8065, ameya.joshi, chinmay.h}@nyu.edu
{baditya, anjana, bkhara, soumiks, baskarg, adarsh}@iastate.edu [*]

Abstract

The paradigm of differentiable programming has significantly enhanced the scope of machine learning via the judicious use of gradient-based optimization. However, standard differentiable programming methods (such as autodiff) typically require the machine learning models to be differentiable, limiting their applicability. Our goal in this paper is to use a new, principled approach to extend gradient-based optimization to functions well modeled by splines, which encompass a large family of piecewise polynomial models. We derive the form of the (weak) Jacobian of such functions and show that it exhibits a block-sparse structure that can be computed implicitly and efficiently. Overall, we show that leveraging this redesigned Jacobian in the form of a differentiable "layer" in predictive models leads to improved performance in diverse applications such as image segmentation, 3D point cloud reconstruction, and finite element analysis. We also open-source the code at https://github.com/idealab-isu/DSA.

1 Introduction

Motivation: Differentiable programming has been a paradigm shift in algorithm design. The main idea is to leverage gradient-based optimization to optimize the parameters of the algorithm, allowing for end-to-end trainable systems (such as deep neural networks) to exploit structure in data and achieve better performance. This approach has found use in a large variety of applications such as scientific computing [Innes, 2020; Innes et al., 2019; Schafer et al., 2020], image processing [Li et al., 2018a], physics engines [Degrave et al., 2017], computational simulations [Alnæs et al., 2015], and graphics [Li et al., 2018b; Chen et al., 2019]. One way to leverage differentiable programming modules is to encode additional structural priors as "layers" in a larger machine learning model. Inherent structural constraints such as monotonicity, or piecewise constancy, are particularly prevalent in applications such as physics simulations, graphics rendering, and network engineering. In such applications, it may be beneficial to build models that obey such priors by design.

Challenges: For differentiable programming to work, all layers within the model must admit simple gradient calculations; however, this poses a major limitation in many settings. For example, consider computer graphics applications for rendering 3D objects [Kindlmann et al., 2003; Gross et al., 1995; Loop and Blinn, 2006]. A common primitive in such cases is a *spline* (or a piecewise polynomial) function which either exactly or approximately interpolates between a discrete set of points to produce a continuous shape or surface. Similar spline (or other piecewise polynomial) approximations arise in

[*†] Equal contribution.

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

partial differential equation (PDE) solvers [Hughes et al., 2005], network flow problems [Balakrishnan and Graves, 1989], and other applications.

For such problems, we would like to compute gradients "through" operations involving spline approximation. However, algorithms for spline approximation often involve discontinuous (or even discrete) co-domains and may introduce undefined (or even zero) gradients. Generally, embedding such functions as layers in a differentiable program, and running automatic differentiation on this program, requires special care. A popular solution is to relax these non-differentiable, discrete components into continuous approximations for which gradients exist. This has led to recent advances in differentiable sorting [Blondel et al., 2020; Cuturi et al., 2019], dynamic programming [Mensch and Blondel, 2018], and optimization [Djolonga and Krause, 2017; Agrawal et al., 2019; Deng et al., 2020].

Our contributions: We propose a principled approach for differentiable programming for spline functions *without* the use of continuous relaxation[2]. For the forward pass, we leverage fast algorithms for computing the optimal projection of any given input onto the space of piecewise polynomial functions. For the backward pass, we leverage a fundamental *locality* property in splines that every piece (or basis function) in the output approximation only interacts with a few other elements. Using this, we derive a weak form of the Jacobian for the spline operation and show that it exhibits a particular block-structured form. While we focus on spline approximation in this paper, our approach can be generalized to any algorithmic module with piecewise smooth outputs. Our specific contributions are as follows:

1. We propose the use of spline function approximations as "layers" in differentiable programs.
2. We derive efficient (nearly-linear time) methods for computing forward and backward passes for various spline approximation problems, showing that the (weak) Jacobian in each case can be represented using a *block sparse* matrix that can be efficiently used for backpropagation.
3. We show applications of our approach in three stylized applications: image segmentation, 3D point cloud reconstruction, and finite element analysis for the solution of partial differential equations.

Related Work Before proceeding, we briefly review related work.

Extensions of autodiff: Automatic differentiation (autodiff) algorithms enable gradient computations over basic algorithmic primitives such as loops, recursion, and branch conditions [Baydin et al., 2018]. However, introducing more complex non-differentiable components requires careful treatment due to undefined or badly behaved gradients. For example, in the case of sorting and ranking operators, it can be shown that the corresponding gradients are either uninformative or downright pathological, and it is imperative the operators obey a 'soft' differentiable form. Cuturi et al. [2019] propose a differentiable *proxy* for sorting based on optimal transport. Blondel et al. [2020] improve this by proposing a more efficient differentiable sorting/ranking operator by appealing to isotonic regression. Berthet et al. [2020] introduce the use of stochastic perturbations to construct smooth approximations to discrete functions, and other researchers have used similar approaches to implement end-to-end trainable top-k ranking systems [Xie et al., 2020; Lee et al., 2020]. Several approaches for enabling autodiff in optimization have also been researched [Pogančić et al., 2020; Amos and Kolter, 2019; Agrawal et al., 2019; Mensch and Blondel, 2018].

Structured priors as neural "layers": As mentioned above, one motivation for our approach arises from the need for enforcing structural priors for scientific computing applications. Encoding non-differentiable priors such as the solutions to specific partial differential equations [Sheriffdeen et al., 2019], geometrical constraints [Joshi et al., 2020; Chen et al., 2019], and spatial consistency measures [Djolonga and Krause, 2017] perform well but typically require massive amounts of structured training examples.

Spline approximation: Non-Uniform Rational B-splines (NURBS) are commonly used for defining spline surfaces for geometric modeling [Piegl and Tiller, 1997]. NURBS surfaces offer a high level of control and versatility; they can also compactly represent the surface geometry. The versatility of NURBS surfaces enables them to represent more complex shapes than Bèzier or B-splines. Several frameworks that leverage deep learning are beginning to use NURBS representations. Minto et al. [2018] use NURBS surfaces fitted over the 3D geometry as an input representation for the object

[2]While tricks such as straight-through gradient estimation [Bengio, 2013] also avoid continuous relaxation, they are heuristic in nature and may be inaccurate for specific problem instances [Yin et al., 2019].

classification task of ModelNet10 and ModelNet40 datasets. Erwinski et al. [2016] presented a neural-network-based contour error prediction method for NURBS paths. Fey et al. [2018] present a new convolution operator based on B-splines for irregular structured and geometric input, e.g., graphs or meshes. Very recently, Sharma et al. [2020] perform point cloud reconstruction to predict a B-spline surface, which is later processed to obtain a complete CAD model with other primitives "stitched" together.

Differentiable PDE solvers: With the advent of deep learning, there has been a recent rise in the development of differentiable programming libraries for physics simulations [Hu et al., 2019; Qiao et al., 2020]. Most often, the physics phenomena are represented using partial differential equations (PDEs) [Sanchez-Gonzalez et al., 2020; Holl et al., 2020]. Considerable effort has gone into designing physics-informed loss functions [Raissi et al., 2019; Raissi and Karniadakis, 2018; Kharazmi et al., 2021] whose optimization leads to desired solutions for PDEs. Due to space limitations, we defer to a detailed survey of this (vast) area by Cai et al. [2021].

2 Differentiable Spline Approximation

We now introduce our framework, Differentiable Spline Approximation (DSA), as an approach to estimate gradients over piecewise polynomial operations. Our main goal will be to estimate easy-to-compute forms of the (weak) Jacobian for several spline approximation problems, enabling their use within backward passes in general differentiable programs.

Setup. We begin with some basic definitions and notation. Let $f \in \mathbb{R}^n$ be a vector where the i^{th} element is denoted as f_i. Let us use $[n] = \{1, 2, \ldots, n\}$ to denote the set of all coordinate indices. For a vector $f \in \mathbb{R}^n$ and an index set $I \subseteq [n]$, let f_I be the restriction of f to I, i.e., for $i \in I$, we have $f_I(i) := f_i$, and $f_I(i) := 0$ for $i \notin I$. Now, consider any fixed partition of $[n]$ into a set of disjoint intervals $\mathcal{I} = \{I_1, \ldots, I_k\}$ where the number of intervals $|\mathcal{I}| = k$. The ℓ_2-norm of f is written as $\|f\|_2 := \sqrt{\sum_{i=1}^n f_i^2}$ while the ℓ_2 distance between f, g is written as $\|f - g\|_2$.

We first define the notion of a *discretized k-spline*. Note that the use of "spline" here is non-standard and somewhat more general than what is typically encountered in the literature. (Indeed, the spline concept used in computer graphics is a special instance of this definition; we explain further below.)

Def. 2.1 (Discretized k-spline)**.** A vector $h \in \mathbb{R}^n$ is called a discretized k-spline with degree d if: (i) there exists a partition of $[n]$ into k disjoint intervals $I_1, \ldots, I_k$; (ii) within each interval I_i, the coefficients of h_j, $j \in I_i$, can be perfectly interpolated by some polynomial function of degree d.

Let us illustrate this by an example. Suppose that $d = 1$ and $k = 5$. Then, h is a discretized k-spline with degree d if, in a "line plot" of the vector h (i.e., we interpolate the 2D points (j, h_j) for all $j \in [n]$), we see up to $k = 5$ distinct linear pieces. A different way to interpret this definition is that we start with a piecewise degree-d polynomial function $H : \mathbb{R} \to \mathbb{R}$ with $k = 5$ pieces (with suitably defined knot points, which are the location of the intervals I), and evaluate H at any n equally spaced points in its domain. This gives us a vector $h \in \mathbb{R}^n$, which we call a discretized k-spline. In contrast with traditional splines, we allow H to be arbitrarily defined *at* the knot points and require no specific continuity or differentiability properties. Therefore, our definition encompasses all standard spline families (including interpolating/approximating splines such as smoothing-, cubic-, and B-splines).

2.1 Spline Approximation

Our focus in this paper is the problem of computing the best possible spline fit to a given set of data points (where both the parameters of the spline as well as the knot vectors are allowed to be variable).

We provide an algebraic interpretation of this problem. For a given vector space $\mathbb{R}^n$, consider S_d^k, the set of all discretized k-splines with degree d. Since (standard) splines are vector spaces for a fixed set of knots, one can easily see that for any fixed partition of $[n]$ into k subsets, the family of discretized k-splines is a $k(d+1)$-dimensional subspace of $\mathbb{R}^n$. Now suppose that the knot indices are allowed to vary. The number of possible partitions is finite (of the order of $\binom{n}{k}$), and therefore the set S_d^k is a *finite union of subspaces*, or a nonlinear submanifold, embedded in $\mathbb{R}^n$.

Therefore, the problem of discretized k-spline approximation can be viewed as an *orthogonal projection* onto this nonlinear manifold. Consider any arbitrary vector $x \in \mathbb{R}^n$ (we can think of (i, x_i)

as a set of n data points to which we are trying to fit a k-spline). Then, the best k-spline fit to x (in the sense of ℓ_2 distance) amounts to solving the optimization problem:

$$F(x) = \arg\min_h \frac{1}{2}\|x - h\|_2^2 = \frac{1}{2}\sum_{i=1}^{n}(x_i - h_i)^2 \;\text{ s.t. } h \in S_d^k \tag{1}$$

This operation resembles standard spline regression. But it is strictly more general since this requires not only optimizing piecewise spline parameters but *also the knot indices*. Crucially, we note that F is both a non-differentiable and a non-convex map. Nevertheless, such an orthogonal projection can be computed in polynomial (in fact, nearly-linear) time [Jagadish et al., 1998; Acharya et al., 2015] using many different techniques, including dynamic programming. This forms the *forward pass* of our DSA "layer".

Our first main conceptual contribution is a formal derivation of the *backward pass* of the orthogonal projection operation. Strictly speaking, the Jacobian is not well-defined due to the non-differentiable nature of the forward pass (owing to the non-differentiability built into the definition of the k-spline). Therefore, we will instead be deriving the so-called "weak" form of the Jacobian (borrowing terminology from Blondel et al. [2020]).

We leverage two properties of the projection operation: (1) the output of the forward pass h corresponds to a partition of $[n]$, that is, each element of h_j corresponds to a *single* interval, I_j, and (2) within each interval, the least-squares operation is continuous and differentiable. The first property ensures that every element x_i contributes to only a single piece in the output h. Given that the sub-functions from the piecewise partitioning function are smooth, we also observe that the size of each block corresponds to the size of the partition, I_i. Using this observation, we get:

Theorem 1. The Jacobian of the operation F with respect to $x \in \mathbb{R}^n$ can be expressed as a *block diagonal* matrix, $\mathbf{J} \in \mathbb{R}^{n\times n}$, whose $(s,t)^{\text{th}}$ entry obeys:

$$\mathbf{J}_x(F(x))(s,t) = \frac{\partial h(x)_s}{\partial x_t} = \begin{cases} \frac{\partial h_{I_i}(x)_s}{\partial x_t} & \text{if } s,t \in I_i \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

As a concrete instantiation of this result, consider the case $d = 0$. This is the case where we wish to best approximate the entries of x with at most k "horizontal" pieces, where the break-points are obtained during the forward pass[3]. Call this approximation h. Then, the Jacobian of h with respect to x forms the block-diagonal matrix $\mathbf{J} \in \mathbb{R}^{n\times n}$:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{J}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{J}_k \end{bmatrix} \tag{3}$$

where all entries of each block, $\mathbf{J}_i \in \mathbb{R}^{|I_i|\times|I_i|}$ are *constant* and equal to $1/|I_i|$, i.e., they are row/column-stochastic. Note that the sparse structure of the Jacobian allows for fast computation and that computing the Jacobian vector product $\mathbf{J}^T\nu$ for any input ν requires $O(n)$ running time. As an additional benefit, the decoupling induced by the partition enables further speed up in computation via parallelization. See the Appendix for proofs, as well as derivations of similar Jacobians for k-spline approximation of any degree $d \geq 1$, and generalization to 2D domains (surface approximation). In Section 3 we demonstrate the utility of this approach for a 2D segmentation (i.e., piecewise constant approximation) problem, similar to the setting studied in Djolonga and Krause [2017].

[3] In the data summarization literature, this class of functions is sometimes called k-histograms [Jagadish et al., 1998]

2.2 Differentiable NURBS

We now switch to a slightly different setting involving a special spline family known as non-uniform rational B-splines (NURBS), which are common in geometric modeling. Mathematically, a NURBS curve is a continuous function $\mathbf{C} : \mathbb{R} \to \mathbb{R}$ defined as follows. Construct any knot vector u (i.e. a non-decreasing sequence of real coordinate values) and fix degree d. Recursively define a sequence of basis functions, $N_i^d : \mathbb{R} \to \mathbb{R}$ computed using the *Cox-de Boor formula*:

$$N_i^d(u) = \frac{u - u_i}{u_{i+d} - u_i} N_i^{d-1}(u) + \frac{u_{i+d+1} - u}{u_{i+d+1} - u_{i+1}} N_{i+1}^{d-1}(u), \; N_i^0(u) = \begin{cases} 1 & \text{if } u_i \le u \le u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

for $d = 1, 2, \ldots$. In the uniform case (where the knots are equally spaced), each N_i^d can be viewed as being generated by recursively convolving a box function with N_i^{d-1}. The non-uniform case cannot be written as a convolution, but the intuition is similar. With these basis functions in hand, the NURBS curve $\mathbf{C}$ is defined as the rational function:

$$\mathbf{C}(u) = \frac{\sum_{i=0}^{n} N_i^d(u) w_i \mathbf{P}_i}{\sum_{i=0}^{n} N_i^d(u) w_i}, \tag{5}$$

where $P_i, \; i = 0, 1, \ldots, t$ are called *control points* and w_i are corresponding non-negative weights. The number of control points is related to the number of knots k and curve degree d as follows: $k = t + d + 1$. For simplicity, assume that all weights are equal to one. The basis functions in NURBS add up to one uniformly for each u (this is called the *partition of unity* property). Therefore:

$$\mathbf{C}(u) = \sum_{i=0}^{t} N_i^d(u) \mathbf{P}_i, \tag{6}$$

In summary, the NURBS curve is parametrically defined via the control points and the knot positions. This discussion is for 1D curves, but an extension to higher-order surfaces is conceptually similar.

Consider implementing NURBS as a differentiable "layer" where the inputs are the knot positions and control points. The forward pass through this layer simply consists of evaluating Equation 6 via the recursive Equation 4, and storing the various basis functions (and their spans) for further use.

However, the backward pass is a bit more complicated, once again due to the *non-differentiable* nature of $\mathbf{C}$. The gradient with respect to the control point coordinates, $\mathbf{P}$ is straightforward since the mapping from $\mathbf{P}$ to $\mathbf{C}$ is linear. However, the gradient with respect to the *knot* positions, u_i, is not well-defined due to the non-differentiable nature of the *base cases* of the recursion (which are box functions specified in terms of u_i). Once again, we see that the non-differentiability of NURBS is built into its very definition, and this affects the numerics.

To resolve this, we propose the following approach to compute an (approximate) Jacobian of $\mathbf{C}$. The main source of the issue is the derivative of the box-car function $N_i^0(u) = \mathbf{1}_{[u_i, u_{i+1})}$ with respect to the knot points, which is not well defined. However, $N_i^0(u)$ can be viewed as the difference between convolutions of the unit step function with δ_{u_i} and $\delta_{u_{i+1}}$, where δ is the Dirac delta defined over the real line. We smoothly approximate the delta function by a Gaussian function with small enough bandwidth hyperparameter σ: $\delta(u_i) \approx g(u) = \exp(-(u - u_i)/2\sigma^2)$. This function is now differentiable with respect to u_i, with $g'(u) = \frac{u - u_i}{\sigma^2} g(u)$. Convolutions and differences are linear, and hence the derivative is the basis function times a multiplicative factor. Finally, a similar approach as the Cox-de Boor recursion (Equation 4) can be used to reconstruct the derivatives for all basis functions of higher order. See Algorithm 1 for pseudocode and the Appendix for details.

Algorithm 1 Backward pass for NURBS Jacobian (for one curve point , $\mathbf{C}(u)$)

$\mathbf{P}'$, $\mathbf{U}'$: gradients of $\mathbf{C}$ w.r.t. $\mathbf{P}$, $\mathbf{U}$
Initialize: $\mathbf{P}', \mathbf{U}' \to 0$
Retrieve u_{span}, N_i^d, $\mathbf{C}(u)$ calculated during forward pass
/* u_{span} is the index of knot position */
/* N_i^d is the basis function of degree d */
/* $\mathbf{C}(u)$ is the evaluated curve point */
for $h = 0 : d + 1$ **do**
    $\mathbf{P}'_{u_{span}+h} = N_h^d$ // easy since $\mathbf{C}$ is a linear function of $\mathbf{P}$.
    $\mathbf{U}'_{u_{span}+h} = N_h^d \, \mathbf{U}_{u_{span}+h}$ // due to Gaussian approximation; see discussion below.

Let us probe the structure of this Jacobian a bit further. Suppose we evaluate the curve $\mathbf{C}$ at n arbitrary domain points. There are slightly less than k control points, and therefore the Jacobian is roughly of size $n \times O(k)$. However, due to the recursive nature of the definition of basis functions, the *span* (or support) of each basis function is small and only touches $d+1$ knots; for example, only 2 knots affect N_i^0, only 3 knots impact N_i^1, and so on. This endows a natural sparse structure on the Jacobian. Moreover, for a fixed order parameter $d+1$, the span is constant [Piegl and Tiller, 1997]; therefore, assuming evenly spaced evaluation points, we have the same number of nonzeros. Therefore, the Jacobian exhibits an interesting *Toeplitz* structure (unlike the block diagonal matrix in the case of Equation 3), thereby enabling efficient evaluation during any gradient calculations. We show below in Section 3 that automatic differentiation using this approach surpasses existing NURBS baselines.

2.3 Differentiable Finite Element PDE Solvers

Next, we see how spline approximations can be used to improve finite element analysis for solving PDEs. Popular recent efforts for solving PDEs using autodiff construct "physics-informed" solvers [Raissi et al., 2019; Raissi and Karniadakis, 2018], while other efforts have been made to utilize variational [Kharazmi et al., 2021] or adjoint-based derivative methods [Holl et al., 2020]. However, these approaches come with challenges while used in conjunction with autodiff packages, and gradient pathologies pose a major barrier [Wang et al., 2020].

Using our principles developed above, we propose an alternative PDE solution approach via *differentiable finite elements*. PDE solvers based on Finite Element Methods (FEM) are ubiquitous, and we provide a very brief primer here. Consider a domain Ω and a differential system of equations:

$$\mathcal{N}[\mathbf{U}(\underline{u})] = F(\underline{u}), \quad \underline{u} \in \Omega, \tag{7}$$

where $\mathcal{N}$ denotes the differential operator and $\mathbf{U} : \Omega \to \mathbb{R}$ is a continuous field variable; it is common to specify additional boundary constraints on $\mathbf{U}$. The *Galerkin method* converts solving for the best possible $\mathbf{U}$ (which is a continuous variable) into a discrete problem by first looking at the weak form: $R(\mathbf{U}) = \int_\Omega V\,[\mathcal{N}(\mathbf{U}) - F]\,d\underline{u}$, where V is called a *test function* (and the weak form may involve some integration by parts), and rewriting this weak form in terms of a finite set of basis coefficients. A typical set of basis functions Φ_j is obtained by (piecewise) concatenation of polynomials, each defined over elements of a given partition of Ω (also called a *mesh*). Commonly used choices include Lagrange polynomials, defined by:

$$p_{i,d}^r(\underline{u}) = \sum_{r=1}^{d} \mathbf{U}_r \prod_{\substack{0 \le m \le d \\ m \ne r}} \frac{\underline{u} - u_m}{u_r - u_m} \text{ s.t. } x_r \in [-1, 1] \tag{8}$$

where $\{u_0, u_1, \ldots, u_d\}$ are a finite set of nodes (akin to control points in our above discussion, except in this case the splines interpolate the control points) and $\mathbf{U}_r$ is the corresponding coefficient. We use this collection of basis functions Φ_j to represent $\mathbf{U}$:

$$\mathbf{U}(\underline{u}) = \sum_{j=1}^{\#\text{nodes}} \Phi_j(\underline{u}) \mathbf{U}_j^d \tag{9}$$

and likewise for V. (The resemblance with Equation 5 above should be clear, and indeed NURBS basis functions could be an alternative choice.) Plugging the discrete coefficient representation $\mathbf{U}^c := \{\mathbf{U}_j^c\}$ into the definition of R, we get a standard Finite Element form,

$$R(\mathbf{U}^c, V^c) = B(\mathbf{U}^c, V^c) - L(V^c) \tag{10}$$

where $B(\mathbf{U}^c, V^c)$ is the discrete form (bilinear for linear operators) that encodes the differential operator and $L(v)$ is a linear functional involving the forcing function. For most PDE operators (including linear elliptic operators), one can form the *energy functional* by using U as the test function:

$$J(\mathbf{U}^c) = \frac{1}{2} B(\mathbf{U}^c, \mathbf{U}^c) - L(\mathbf{U}^c). \tag{11}$$

Optimization of this energy functional can now be performed using gradient-based iterations evaluated by automatic differentiation. This is a powerful approach since formal techniques exist (e.g., Galerkin Least Squares Bochev and Gunzburger [2009]) that reformulate weak forms of PDEs into equivalent energy functionals. The key aspect to note here is that differentiating "through" the differential operator $\mathcal{N}$ (embedded within B) requires derivative computations of the piecewise polynomial basis functions Φ_js, and therefore our techniques developed above are applicable.

3 Experiments

We have implemented the DSA framework (and its different applications provided below) by extending `autograd` functions in Pytorch. We also provide the capability to run the code using CUDA for GPU support. All the experiments were performed using a local cluster with 6 compute nodes and each node having 2 GPUs (Tesla V100s with 32GB GPU memory). All training was done using a single GPU. We summarize all our experiments in Table 1. Each experiment shown below is performed multiple times with different random seeds, and the average value with error bars is provided. Due to limited space, we provide three interesting applications of spline approximations here (see Appendix for additional examples). We have also open-sourced the code at `https://github.com/idealab-isu/DSA`.

Image segmentation: We begin with implementing a 2D piecewise constant splines regression approach for the image segmentation problem using a UNet [Ronneberger et al., 2015]. For differentiation, we use the formulation of splines discussed in Section 2. We analyze the efficacy of our approach by adding a piecewise constant DSA layer as the final layer of our network (M_{DSA}). We compare this approach with the baseline model without the piecewise constant layer (M_{baseline}).

We train two models (M_{DSA}, M_{baseline}) on two different segmentation tasks: the Weizmann horse dataset [Borenstein and Ullman, 2004] and the Broad Bioimage Benchmark Collection dataset [Ljosa et al., 2012] (publicly available under Creative Commons License). We split both the Weizmann horse and Broad Bioimage Benchmark Collection datasets into train and test with 85% and 15% of the dataset. We use binary cross-entropy error between the ground truth and the predicted segmentation map. We use the same architecture and hyper-parameters for both models (see Appendix for details.)

We observe that our DSA layer provides more consistent segmentation maps and higher Jaccard scores than the baseline model; see Figure 1. For the Weizmann horse dataset, M_{conn} enforces the connectivity of the segmented objects while also limiting noise in the segmentation map. In the cell

Table 1: ***Summary of Experiments:*** *We present three experiments in this paper with diverse applications, model architectures, basis functions, and degree of splines.*

Application	Input	Output	Architecture	Spline Function	Degree
Image Segmentation	Image	2D piecewise constant knot partitions	U-Net	Box-Car functions	0
Point cloud reconstruction	3D point cloud	3D control points and rational weights	Dynamic Graph CNN	BSpline polynomials	3
PDE-based surrogate physics	2D Mesh (regular grid)	2D physics field on the mesh grid	U-Net	Lagrange polynomials	1,2,3

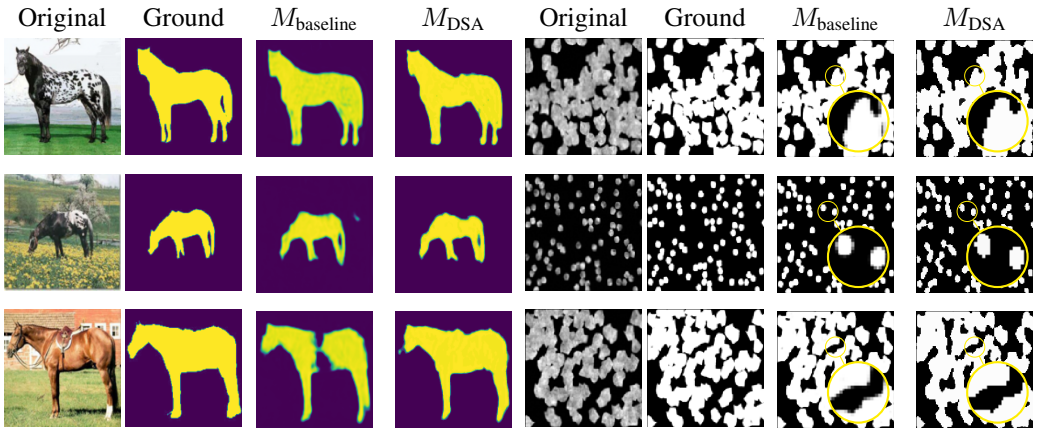

Figure 1: ***Segmentation results.*** *The two models, M_{DSA} and $M_{baseline}$ were trained with and without the DSA layer, respectively. Note that M_{DSA} generates better segmentation masks with fewer holes and enforced connectivity. Note the sharper edges compared to the standard segmentation results. Additional figures are in the Appendix.*

Table 2: ***Results for the horse and cell segmentation dataset:*** *Jaccard scores for the baseline and connected component models for the cell and horse segmentation task. From independent three runs with random seeds and the table reports mean and standard deviation. As the objects of interest (piecewise constant components) are smaller, the model with the DSA layer learns a better representation. Predictions are thresholded at* 0.5.

Dataset	Baseline (M_{baseline})	Baseline + DSA (M_{DSA})
Weizmann Horse [Borenstein and Ullman]	72.06 ± 0.60	$\mathbf{73.13 \pm 0.31}$
Broad Bioimage Benchmark [Ljosa et al.]	79.34 ± 0.43	$\mathbf{81.56 \pm 0.24}$

segmentation task, we note that the number of segments is high while the objects are small. Since the size of the components is small, our DSA layer Jacobian exhibits substantial differences from the commensurate identity gradient for the baseline models. Table 2 also shows the further improvement in Jaccard score on cell segmentation tasks over the Weizmann horse dataset.

3D point cloud reconstruction using NURBS: Next, we provide results for two experiments using DSA with NURBS discussed in Section 2.2. The first application is surface fitting for a complex benchmark surface represented by a mesh of surface points obtained by evaluating the benchmark test function at these points. We use Bukin function N.6 (publicly available here) for generating a grid of 256×256 points as shown on the left of Figure 2. For fitting a NURBS surface from the defined target point cloud, we initialize a uniform clamped knot vector for a cubic basis function and random control points of size 8×8. Using DSA, we evaluate the NURBS surface for a uniform grid of 256×256 parametric points. We now evaluate the surface and use mean squared error for fitting the surface point cloud using NURBS. We consider two scenarios: (i) we do not update the knot vectors (i.e., no reparameterization), and (ii) we compute the gradients for the knot vectors and allow for reparameterization (i.e., change of knot locations). We provide the comparison of these scenarios in Table 3. We see that the reparameterization helps in reducing the error in fit by half. Also, we notice that the density of points evaluated has a very minimal impact on the performance (see more details in Appendix). Visually, in Figure 2, we see that two knots in the "v" direction come close to each other around 0.06, enabling a sharp edge in the evaluated surface.

The next experiment we present involves surface reconstruction from point clouds using a graph convolutional neural network and DSA for unsupervised training. We use the SplineNet method proposed by Sharma et al. [2020] to be the baseline for point cloud reconstruction using splines. SplineNet uses a dynamic graph convolutional neural network (DGCNN) to predict the control points for a spline surface. The authors use a supervised control point loss to perform the training and include regularizations such as the Laplacian loss and a patch distance (using Chamfer distance) loss. Instead, we perform this training in an unsupervised manner by not using the control points prediction loss and only using DSA to evaluate the surface and then apply regularization of minimizing the Laplacian of the surface. Since we can train this unsupervised, we can even use an arbitrary number of control points and are not restricted to the target control points.

For a fair comparison, we use the same network, dataset, and hyperparameters as Sharma et al. [2020] and change the loss functions by removing the control point regression loss. For comparison, we compute the chamfer distance between the input point cloud and the NURBS surface fit by the DGCNN model (M_{DSA}) (see Appendix for details of training). We use the Spline Dataset, which is

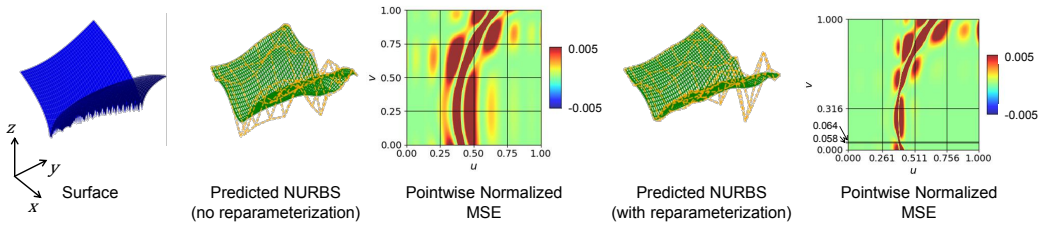

Figure 2: ***NURBS surface fitting results:*** *Surface fitting to point cloud generated using the Bukin's function N.6 given by* $z = 100\sqrt{|y - 0.01x^2|} + 0.01|x + 10|; -15 < x < -5, -3 < y < 3$. *The center image shows the surface fit obtained without reparameterization of the knots. We obtain better fit by reparameterizing the knots.*

Table 3: ***NURBS surface fitting results:*** *Comparison of mean squared error between the target surface point cloud and the surface generated using DSA with and without reparameterization.*

Number of Points	M_{DSA}(without reparameterization)	M_{DSA}(with reparameterization)
128×128	19.83 ± 0.001	$\mathbf{8.25 \pm 0.01}$
256×256	19.85 ± 0.001	$\mathbf{8.23 \pm 0.02}$

Table 4: ***Point-cloud reconstruction results:*** *Comparison between the model proposed by Sharma et al. [2020] and its extension using DSA (with different number of control points). We compare the two-sided chamfer distance (scaled by 100) between the input point cloud and the fitted surface.*

Experiment	$M_{baseline}$ (20×20)	M_{DSA} (20×20)	M_{DSA} (5×5)	M_{DSA} (4×4)
Chamfer Distance	1.18 ± 0.10	0.03 ± 0.02	0.14 ± 0.07	$\mathbf{0.02 \pm 0.01}$

a subset of surfaces extracted from the ABC dataset (available for public use under this license). In Table 4, we provide a comparison of chamfer distance obtained between the predicted surface points from splines and the input point cloud for the test dataset. In our experiments, we observe that we get significantly better performance with fewer control points. This is because most of the surfaces in the dataset are simple curved surfaces that can be easily fit with fewer control points.

PDE based surrogate physics priors: Finally, we leverage DSA in the context of solving PDEs as a prior. In particular, we consider the Poisson equation solved for u:

$$-\underline{\nabla} \cdot (\nu(\mathbf{x})\underline{\nabla} u) = f(\mathbf{x}) \text{ in } D \tag{12}$$

$$u|_{\partial D} = 0 \tag{13}$$

where $D = [0, 1]^2$, a 2D square domain, ν is the *diffusivity* and f is the forcing function. We consider two experiments here: (1) validation of our approach with an analytically known solution, and (2) extending this to learn the solutions for the parametric Poisson equation parameterized using ν.

For the first experiment, we set ν to 1 and the forcing $f = f(\underline{x}) = f(x, y) = 2\pi^2 \sin(\pi x) \sin(\pi y)$, and minimize the residual using the approach described in Section 2.3. We know that for this PDE and the conditions provided, the exact solution is given by $u_{ex}(x, y) = \sin(\pi x) \sin(\pi y)$. We compare our results (u_{DSA}) with the exact solution u_{ex}. Also, we perform this experiment with Lagrange polynomials of different degrees. Further, we compare our results with results obtained using PINNs [Raissi et al., 2019]. We obtain significantly better performance (lesser ℓ_2-error by an order of magnitude) compared to PINNs, owing to more accurate gradients computed using our DSA approach. The performance improvement with increase in degree of polynomial in lower resolutions is more pronounced than at higher resolutions.

Next, we present results for training a deep learning network with a prior for solving a *parametric* Poisson's equation. The input to the network are different diffusivity maps ν sampled from

$$\nu(\mathbf{x}; \omega) = \exp\left(\sum_{i=1}^{m} \omega_i \lambda_i \xi_i(x) \eta_i(y)\right) \tag{14}$$

where ω_i is an m-dimensional parameter, λ is a vector of real numbers with monotonically decreasing values arranged in order; and ξ and η are functions of x and y respectively. We take $m = 4$, $\omega = [-3, 3]^4$ and $\lambda_i = \frac{1}{(1+0.25a_i^2)}$, where $\mathbf{a} = (1.72, 4.05, 6.85, 9.82)$. Also $\xi_i(x) = \frac{a_i}{2}\cos(a_i x) + \sin(a_i x)$ and $\eta(y) = \frac{a_i}{2}\cos(a_i y) + \sin(a_i y)$. We generate several diffusivity maps by sampling this function with different values of ω.

Table 5: ***Quantitative comparison of Solving PDEs:*** *L_2 Norm between the analytical exact solution u_{ex} and predicted u using PINNs [Raissi et al., 2019] and DSA with different degrees of the Lagrange polynomials.*

Model		$PINN$	$DSA\,(d=1)$	$DSA\,(d=2)$	$DSA\,(d=3)$
L_2 Norm	128×128	3.72 ± 0.20 E-4	3.32 ± 0.05 E-5	$\mathbf{2.16 \pm 0.04}$ **E-5**	2.37 ± 0.10 E-5
	256×256	2.63 ± 0.20 E-4	$\mathbf{2.57 \pm 0.01}$ **E-5**	2.79 ± 0.20 E-5	2.59 ± 0.10 E-5

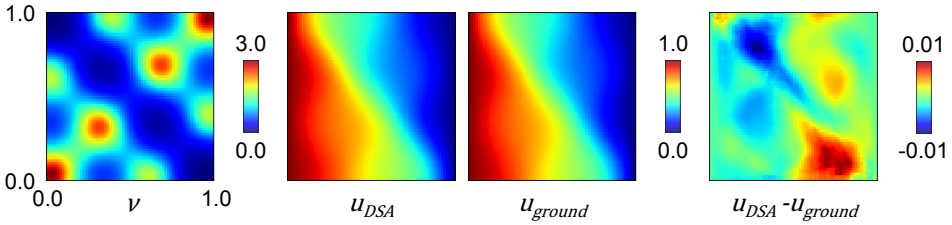

Figure 3: ***Learning a parametric family of PDE solutions:*** *Poisson's equation with log permeability coefficients* $\omega = (-0.26, -0.77, -0.37, -0.92)$ *in the Poisson's equation.*

We use a UNet [Ronneberger et al., 2015] that takes these diffusivity maps and predicts the solution u, which is further optimized with the residual minimizing prior to the Poisson's equation. Thus, we obtain a trained neural network that predicts the solution field u for any unknown diffusivity maps from the data distribution. We provide the predicted result along with its comparison with traditional numerical FEM results in Figure 3. Visually, we see both the predicted solution field map (u_{DSA}) and the actual solution field (u_{ground}) obtained using traditional numerical methods match each other. The right most image shows the difference between both with the maximum deviation to be 0.01, showing the accuracy of our (easy-to-implement) DSA-based FEM solver.

4 Broader Impact and Discussion

We introduce a principled approach to estimate gradients for spline approximations. Specifically, we derive the (weak) Jacobian in the form of a block-sparse matrix based on the partitions generated by any spline approximation algorithm (which serves as the forward pass). The block structure allows for fast computation of the backward pass, thereby extending the application of differentiable programs (such as deep neural networks) to tasks involving splines. Our methods show superior performance than the state-of-the-art curve fitting methods by reducing the chamfer distance by an order of magnitude and the mean squared error in the case of surface fitting by a factor of two. Further, with the application of our methods in finite element analysis, we show significantly better performance than state-of-the-art physics-informed neural networks.

Our method is quite generic and may impact applications such as computer graphics, physics simulations, and engineering design. Care should be taken to ensure that these applications are deployed responsibly. Future works include further algorithmic understanding of the inductive bias encoded by DSA layers and dealing with splines having a dynamically chosen number of parameters (control points and knots).

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCF-2005804, LEAP-HI:2053760, CMMI:1644441, CPS-FRONTIER:1954556, USDA-NIFA:2021-67021-35329 and ARPA-E DIFFERENTIATE:DE-AR0001215. Any information provided and opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of, nor any endorsements by, the funding agencies.

References

Michael J. Innes. Algorithmic differentiation. In *Machine Learning and Systems*, pages 1–12, 2020.

Mike Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and Will Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. *ArXiv*, abs/1907.07587, 2019.

F. Schafer, M. Kloc, C. Bruder, and N. Lorch. A differentiable programming method for quantum control. *ArXiv*, 2020.

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics*, 37(4):1–13, 2018a.

J. Degrave, Michiel Hermans, J. Dambre, and F. Wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers Neurorobotics*, 13, 2017.

Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable Monte-Carlo ray tracing through edge sampling. *ACM Transactions on Graphics*, 37(6):1–11, 2018b.

Wenzheng Chen, Huan Ling, Jun Gao, Edward Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. Learning to predict 3D objects with an interpolation-based differentiable renderer. In *Advances in Neural Information Processing Systems*, volume 32, pages 1–11, 2019.

Gordon Kindlmann, Ross Whitaker, Tolga Tasdizen, and Torsten Moller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *IEEE Visualization, 2003. VIS 2003.*, pages 513–520. IEEE, 2003.

Markus H Gross, Lars Lippert, A Dreger, and R Koch. A new method to approximate the volume-rendering equation using wavelet bases and piecewise polynomials. *Computers & Graphics*, 19(1):47–62, 1995.

Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. In *SIGGRAPH*, pages 664–670. ACM, 2006.

Thomas JR Hughes, John A Cottrell, and Yuri Bazilevs. Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Comp. methods in Applied Mechanics and Engineering*, 194(39-41): 4135–4195, 2005.

Anantharam Balakrishnan and Stephen C Graves. A composite algorithm for a concave-cost network flow problem. *Networks*, 19(2):175–202, 1989.

Mathieu Blondel, O. Teboul, Quentin Berthet, and Josip Djolonga. Fast differentiable sorting and ranking. *ArXiv*, abs/2002.08871, 2020.

Marco Cuturi, O. Teboul, and Jean-Philippe Vert. Differentiable ranking and sorting using optimal transport. In *Neural Information Processing Systems*, 2019.

A. Mensch and Mathieu Blondel. Differentiable dynamic programming for structured prediction and attention. *ArXiv*, abs/1802.03676, 2018.

Josip Djolonga and Andreas Krause. Differentiable learning of submodular models. In *Adv. Neural Inf. Proc. Sys. (NeurIPS)*, pages 1013–1023, 2017.

A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. In *Adv. Neural Inf. Proc. Sys. (NeurIPS)*, 2019.

Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton, and Andrea Tagliasacchi. CvxNet: Learnable convex decomposition. In *IEEE Conf. Comp. Vision and Pattern Recog*. IEEE, 2020.

Yoshua Bengio. Estimating or propagating gradients through stochastic neurons. *ArXiv*, abs/1305.2982, 2013.

Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2019.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Machine Learning Research*, 18(153):1–43, 2018. URL http://jmlr.org/papers/v18/17-468.html.

Quentin Berthet, Mathieu Blondel, O. Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis R. Bach. Learning with differentiable perturbed optimizers. *ArXiv*, abs/2002.08676, 2020.

Yujia Xie, Hanjun Dai, M. Chen, Bo Dai, Tuo Zhao, H. Zha, Wei Wei, and T. Pfister. Differentiable top-k operator with optimal transport. *ArXiv*, abs/2002.06504, 2020.

Hyunsung Lee, Yeongjae Jang, Jaekwang Kim, and Honguk Woo. A differentiable ranking metric using relaxed sorting opeartion for top-k recommender systems. *ArXiv*, abs/2008.13141, 2020.

Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2020. URL https://openreview.net/forum?id=BkevoJSYPB.

Brandon Amos and J. Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. *ArXiv*, 1703.00443, 2019.

Sheroze Sheriffdeen, J. Ragusa, J. Morel, M. Adams, and T. Bui-Thanh. Accelerating PDE-constrained inverse solutions with deep learning and reduced order models. *ArXiv*, abs/1912.08864, 2019.

Ameya Joshi, Minsu Cho, Viraj Shah, B. Pokuri, Soumik Sarkar, Baskar Ganapathysubramanian, and Chinmay Hegde. InvNet: Encoding geometric and statistical invariances in deep generative models. In *Association for the Advancement of Artificial Intelligence Conference*, pages 1–8, 2020.

Les Piegl and Wayne Tiller. *The NURBS Book (2nd Ed.)*. Springer-Verlag, Berlin, Heidelberg, 1997. ISBN 3540615458.

Ludovico Minto, Pietro Zanuttigh, and Giampaolo Pagnutti. Deep learning for 3D shape classification based on volumetric density and surface approximation clues. In *VISIGRAPP (5: VISAPP)*, pages 317–324, 2018.

Krystian Erwinski, Marcin Paprocki, Andrzej Wawrzak, and Lech M Grzesiak. Neural network contour error predictor in CNC control systems. In *2016 21st International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 537–542. IEEE, 2016.

Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. SplineCNN: Fast geometric deep learning with continuous B-spline kernels. In *Conference on Computer Vision and Pattern Recognition*, pages 869–877, 2018.

Gopal Sharma, Difan Liu, Subhransu Maji, Evangelos Kalogerakis, Siddhartha Chaudhuri, and Radomír Měch. ParSeNet: A parametric surface fitting network for 3D point clouds, 2020.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

Yi-Ling Qiao, Junbang Liang, Vladlen Koltun, and Ming C Lin. Scalable differentiable physics for learning and control. *arXiv preprint arXiv:2007.02168*, 2020.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468. PMLR, 2020.

Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. *arXiv preprint arXiv:2001.07457*, 2020.

M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686 – 707, 2019. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2018.10.045. URL http://www.sciencedirect.com/science/article/pii/S0021999118307125.

Maziar Raissi and George Em Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, 2018.

Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, 374:113547, 2021.

Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (pinns) for fluid mechanics: A review. *arXiv preprint arXiv:2105.09506*, 2021.

H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *Proc. of Int. Conference on Very Large Data Bases (VLDB)*, 1998.

Jayadev Acharya, Ilias Diakonikolas, Chinmay Hegde, Jerry Zheng Li, and Ludwig Schmidt. Fast and near-optimal algorithms for approximating distributions by histograms. In *Proc. ACM SIGMOD-SIGACT-SIGAI Symp. on Principles of Database Systems*, 2015.

Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient pathologies in physics-informed neural networks. *arXiv preprint arXiv:2001.04536*, 2020.

Pavel B Bochev and Max D Gunzburger. *Least-squares finite element methods*, volume 166. Springer Science & Business Media, 2009.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

Eran Borenstein and Shimon Ullman. Learning to segment. In *Euro. Conf. Comp. Vision*, pages 315–328. Springer, 2004.

Vebjorn Ljosa, Katherine L Sokolnicki, and Anne E Carpenter. Annotated high-throughput microscopy image sets for validation. *Nature methods*, 9(7):637–637, 2012.

Sudipto Guha, Nick Koudas, and Kyuseok Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Trans. on Database Systems (TODS)*, 31(1):396–438, 2006.

Kesheng Wu, Ekow Otoo, and Arie Shoshani. Optimizing connected component labeling algorithms. In *Medical Imaging 2005: Image Processing*, volume 5747, pages 1965–1976. International Society for Optics and Photonics, 2005.

Adarsh Krishnamurthy, Rahul Khardekar, Sara McMains, Kirk Haller, and Gershon Elber. Performing efficient NURBS modeling operations on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15 (4):530–543, 2009.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] In Introduction, we provide specific contributions we make in this paper.
   (b) Did you describe the limitations of your work? [Yes] In Section 4.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We have reviewed the guidelines and we conform to them.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] In Section 2. We provide the definitions and formulations and assumptions.
   (b) Did you include complete proofs of all theoretical results? [Yes] We include it in Supplement.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Included in Section 3.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Partial details are included in Section 3. Rest of them in Appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] All the tables have error bars.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] In the beginning of Section 3.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] All the codes and datasets are hyperlinked.
   (b) Did you mention the license of the assets? [Yes] We provide the license information in parenthesis for each case, except for one case where the dataset is publicly available, but with unclear license information.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We will make the code public.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] We do not use any such information.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

A Proofs and derivations

Theorem 1. The Jacobian of the operation F with respect to $x \in \mathbb{R}^n$ can be expressed as a *block diagonal* matrix, $\mathbf{J} \in \mathbb{R}^{n \times n}$, whose $(s,t)^{\text{th}}$ entry obeys:

$$\mathbf{J}_x(F(x))(s,t) = \frac{\partial h(x)_s}{\partial x_t} = \begin{cases} \frac{\partial h_{I_i}(x)_s}{\partial x_t} & \text{if } s,t \in I_i \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

Proof. The proof follows similar arguments as in Proposition 4 from Blondel et al. [2020].

Let $\mathcal{I} = \{I_1, I_2, \cdots, I_k\}$ be k partitions induced by some $H : \mathbb{R} \to \mathbb{R}$ for some input, $\mathbf{x} \in \mathbb{R}^n$ and $h \in \mathbb{R}^n$ be a vector from n equally spaced evaluated H in its domain. Then, each element, x_i uniquely belongs to some partition I_r.

Now,

$$\begin{aligned} \mathbf{J}_x(F(x))(s,t) &= \frac{\partial \sum_{j=1}^{k} h(x)_s \odot \mathbb{1}(s \in I_j)}{\partial x_t} \\ &= \begin{cases} \frac{\partial h(x)_s}{\partial x_t} & \text{if } \ s,t \in I_r \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Note that this is a block-diagonal matrix with each block being $|I_r| \times |I_r|$, giving us the required statement. □

B Application of DSA to piecewise polynomial regression

1D piecewise constant regression: We first provide the notations we provided in Section 2.

Let $f \in \mathbb{R}^n$ be a vector where the i^{th} element is denoted as f_i. Let us use $[n] = \{1, 2, \ldots, n\}$ to denote the set of all coordinate indices. For a vector $f \in \mathbb{R}^n$ and an index set $I \subseteq [n]$, let f_I be the restriction of f to I, i.e., for $i \in I$, we have $f_I(i) := f_i$, and $f_I(i) := 0$ for $i \notin I$. Now, consider any fixed partition of $[n]$ into a set of disjoint intervals $\mathcal{I} = \{I_1, \ldots, I_k\}$ where the number of intervals $|\mathcal{I}| = k$. The ℓ_2-norm of f is written as $\|f\|_2 := \sqrt{\sum_{i=1}^{n} f_i^2}$ while the ℓ_2 distance between f, g is written as $\|f - g\|_2$. Finally, $\mathbb{1}_I \in \{0,1\}^n$ is a indicator vector where for $i \in I$, $\mathbb{1}_I(i) = 1$ and for $i \notin I$, $\mathbb{1}_I(i) = 0$.

We consider the case of k-piecewise regression in 1D, where we can use any algorithm to approximate a given input vector with a fixed number of piecewise polynomial functions. The simplest example is that of k-piecewise *constant* regression, where a given input vector is approximated by a set of constant segments.

Formally, consider a piecewise constant function $H : \mathbb{R} \to \mathbb{R}$ with k pieces. Similar to spline, we evaluate H at any n equally spaced points in its domain. This gives us a vector $h \in \mathbb{R}^n$, which we call a k-piecewise constant vector. Since the best (in terms of ℓ_2-norm) constant approximation to a function is its mean, a k-piecewise constant function approximation can be reparameterized over the collection of all disjoint intervals $\mathcal{I} = \{I_1, \ldots, I_k\}$ of $[n]$ such that given $\mathbf{x}$:

$$\min_{I_1,\ldots,I_k} \sum_{i=1}^{n} \sum_{j=1}^{k} (h_{I_j}(i) - x_i)^2 = \min_{I_1,\ldots,I_k} \sum_{j=1}^{k} \sum_{i \in I_j} (\frac{1}{|I_j|} \sum_{l \in I_j} x_l - x_i)^2 \tag{16}$$

We assume an optimal H (parameterized by $\{I_i\}$ that can be obtained using many existing methods (a classical approach by dynamic programming [Jagadish et al., 1998]). The running time of such approaches is typically $O(nk)$, which is constant for fixed k; see Acharya et al. [2015] for a more detailed treatment.

Using Theorem 1, the Jacobian of the output k-histogram with respect to $\mathbf{x}$ assumes the following form:

$$\frac{\partial h}{\partial x_i} = \frac{\partial}{\partial x_i}\sum_{j=1}^{k}(\frac{1}{|I_j|}\sum_{l\in I_j} x_l)) = \frac{\partial}{\partial x_i}\sum_{j=1}^{k}(\frac{1}{|I_j|}(\sum_{l\in I_j} x_l)\mathbb{1}_{I_j}) \tag{17}$$

$$= \sum_{j=1}^{k}\frac{\partial}{\partial x_i}\frac{1}{|I_j|}(\sum_{l\in I_j}\mathbb{1}_{I_j}) = \frac{1}{|I_j|}\mathbb{1}_{I_j} \tag{18}$$

Therefore, the Jacobian of h with respect to $\mathbf{x}$ forms the block-diagonal matrix $\mathbf{J} \in \mathbb{R}^{n\times n}$:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{J}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{J}_k \end{bmatrix}$$

where all entries of $\mathbf{J}_i \in \mathbb{R}^{|I_i|\times|I_i|}$ equal to $1/|I_i|$. Note here that the sparse structure of the Jacobian allows for fast computation, and it can be easily seen that computing the Jacobian vector product $\mathbf{J}^T\nu$ for any input ν requires $O(n)$ running time. As an additional benefit, the decoupling induced by the partition enables further speed up in computation via parallelization.

Generalization to 1D piecewise polynomial fitting: We now derive differentiable forms of generalized piecewise d-polynomial regression, which is used in applications such as spline fittings.

As before, $H : \mathbb{R} \to \mathbb{R}$ is any algorithm to compute the k-piecewise d polynomial approximation of an input vector $\mathbf{x} \in \mathbb{R}^d$ that outputs partition $\mathcal{I} = \{I_1, \dots, I_k\}$. Similarly, the function H gives us a vector $h \in \mathbb{R}^n$, a k-piecewise polynomial vector. Then, for each partition, we are required to solve a d-degree polynomial regressions. Generally, the polynomial regression problem is simplified to linear regression by leveraging a Vandermonde matrix. We get a similar closed-form expression for the coefficient as in Section 2.2.

Assume that for partition I_j, the input indices $t_{I_j}(i)$ is i$^{\text{th}}$ element in an index vector corresponding to the I_j partition. Then, the input indices $t_{I_j}(i)$ are represented as a Vandermonde matrix, $\mathbf{V}_{I_j}$:

$$\mathbf{V}_{I_j} = \begin{bmatrix} 1 & t_{I_j}(1) & t_{I_j}(1)^2 & \cdots & t_{I_j}(1)^d \\ 1 & t_{I_j}(2) & t_{I_j}(2)^2 & \cdots & t_{I_j}(2)^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_{I_j}(|I_j|) & t_{I_j}(|I_j|)^2 & \cdots & t_{I_j}(|I_j|)^d \end{bmatrix}.$$

It can be shown that the optimal polynomial coefficient α_{I_j} corresponding to the partition (or disjoint interval) I_j have the following closed form:

$$\alpha_{I_j} = (\mathbf{V}_{I_j}^T\mathbf{V}_{I_j})^{-1}\mathbf{V}_{I_j}^T\mathbf{x}_{I_j},$$

where $\mathbf{x}_{I_j} \in \mathbb{R}^{|I_j|}$ is a vector $\mathbf{x}$ length of $|I_j|$ corresponding to the I_j partition such that $\mathbf{x}_{I_j}(i) = x_i$ if $i \in I_j$ and undefined if $i \notin I_j$. This can be computed in $O(knd^w)$ time where w is the matrix-multiplication exponent [Guha et al., 2006]. Then using Theorem 1 and the gradient for polynomial regression, the Jacobian of h_{I_j} with respect to $\mathbf{x}$ forms a blockwise sparse matrix:

$$\begin{aligned}\frac{\partial h_{I_j}(s)}{\partial x_l} &= \frac{\partial}{\partial x_l}(\langle\alpha_{I_j}, [\mathbf{V}_{I_j}^T]_s\rangle) = \frac{\partial}{\partial x_l}(\langle(\mathbf{V}_{I_j}^T\mathbf{V}_{I_j})^{-1}\mathbf{V}_{I_j}^T\mathbf{x}_{I_j}, [\mathbf{V}_{I_j}^T]_s\rangle) \\ &= \frac{\partial}{\partial x_l}[\mathbf{V}_{I_j}^T]_s^T(\mathbf{V}_{I_j}^T\mathbf{V}_{I_j})^{-1}\mathbf{V}_{I_j}^T\mathbf{x}_{I_j} \\ &= \begin{cases} \left[\mathbf{V}_{I_j}(\mathbf{V}_{I_j}^T\mathbf{V}_{I_j})^{-1}[\mathbf{V}_{I_j}^T])_s\right]_l & \text{if } l, s \in I_j \\ 0 & \text{otherwise.} \end{cases}\end{aligned}$$

The two main takeaways here are as follows: (1) $\mathbf{V}_{I_i}$ can be precomputed for all possible $n-1$ partition sizes, thus allowing for fast ($O(n)$) computation of Jacobian-vector products; and (2) an added flexibility is that we can independently control the degree of the polynomial used in each of the partitions. The second advantage could be very useful for heterogeneous data as well as considering boundary cases in data streams.

B.1 2D piecewise constant functions

Our 1D piecewise spline approximation can be (heuristically) extended to 2D data. We provide detailed descriptions. We consider the problem of image segmentation, which can be viewed as representing the domain of an image into a disjoint union of subsets. Neural-network-based segmentation involves training a model (deep or otherwise) to map the input image to a segmentation map, which is a piecewise constant spline function. However, standard neural models trained in a supervised manner with image-segmentation map pairs would generate pixel-wise predictions, leading to disconnected regions (or holes) as predictions. We leverage our approach to enforce deep models to predict piecewise constant segmentation maps. In case of 2D images, note that we do not have a standard primitive (for piecewise constant fitting) to serve as the forward pass. Instead, we leverage connected-component algorithms (such as Hoshen-Kopelman, or other, techniques [Wu et al., 2005]) to produce a partition, and the predicted output is a piecewise constant image with values representing the mean of input pixels in the corresponding piece. For the backward pass, we use a tensor generalization of the block Jacobian where each partition is now represented as a channel which is only non-zero in the positions corresponding to the channel. Formally, if the image $\mathbf{x} \in \mathbb{R}^n$ is represented as the union of k partitions, $h = \bigcup_{i=1}^{k} I_i$, the Jacobian, $\mathbf{J_x} = \partial h / \partial \mathbf{x} \in \mathbb{R}^{n \times n}$ and,

$$\mathbf{J_x}(F(x))(s,t) = \begin{cases} \frac{\partial h(x)_s}{\partial x_t} = \frac{1}{|I_i|} & \text{if } s,t \in I_i, \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

Note that I_i here no longer correspond to single blocks in the Jacobian. Here, they will reflect the positions of pixels associated with the various components. However, the Jacobian is still sparsely structured, enabling fast vector operations.

C Implementing DSA with NURBS

C.1 Backward evaluation for NURBS surface

In a modular machine learning system, each computational layer requires the gradient of a loss function with respect to the output tensor for the backward computation or the backpropagation. For our NURBS evaluation layer this corresponds to $\partial \mathcal{L} / \partial \mathcal{S}$. As an output to the backward pass, we need to provide $\partial \mathcal{L} / \partial \Psi$. While we represent $\mathcal{S}$ for the boundary surface, computationally, we only compute $\mathbf{S}$ (the set of surface points evaluated from $\mathcal{S}$). Therefore, we would be using the notation of $\partial \mathbf{S}$ instead of $\partial \mathcal{S}$ to represent the gradients with respect to the boundary surface. Here, we assume that with increasing the number of evaluated points, $\partial \mathbf{S}$ will asymptotically converge to $\partial \mathcal{S}$. Now, we explain the computation of $\partial \mathbf{S} / \partial \Psi$ in order to compute $\partial \mathcal{L} / \partial \Psi$ using the chain rule. To explain the implementation of the backward algorithm, we first explain the NURBS derivatives for a given surface point with respect to the different NURBS parameters.

C.2 NURBS derivatives

We rewrite the NURBS formulation as follows:

$$\mathbf{S}(u,v) = \frac{\mathbf{NR}(u,v)}{w(u,v)} \tag{20}$$

where,

$$\mathbf{NR}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij} \mathbf{P}_{ij}$$

$$w(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij}$$

For the forward evaluation of $\mathbf{S}(u,v) = \mathbf{f}\left(\mathbf{P}, \mathbf{U}, \mathbf{V}, \mathbf{W}\right)$, we can define four derivatives for a given surface evaluation point: $\mathbf{S}_{,u} := \partial \mathbf{S}(u,v) / \partial u$, $\mathbf{S}_{,v} := \partial \mathbf{S}(u,v) / \partial v$, $\mathbf{S}_{,\mathbf{P}} := \partial \mathbf{S}(u,v) / \partial \mathbf{P}$, and $\mathbf{S}_{,\mathbf{W}} := \partial \mathbf{S}(u,v) / \partial \mathbf{W}$. Note that, $\mathbf{S}_{,\mathbf{P}}$ and $\mathbf{S}_{,\mathbf{W}}$ are represented as a vector of gradients $\{\mathbf{S}_{,P_{ij}} \forall P_{ij} \in \mathbf{P}\}$ and $\{\mathbf{S}_{w_{ij}} \forall w_{ij} \in \mathbf{W}\}$. Now, we show the mathematical form of each of these four derivatives. The first

derivative is traditionally known as the parametric surface derivative, $\mathbf{S}_{,u}$. Here, $N^p_{i,u}(u)$ refers to the derivative of basis functions with respect to u.

$$\mathbf{S}_{,u}(u,v) = \frac{\mathbf{NR}_{,u}(u,v)w(u,v) - \mathbf{NR}(u,v)w_{,u}(u,v)}{w(u,v)^2} \tag{21}$$

where,

$$\mathbf{NR}_{,u}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N^p_{i,u}(u)N^q_j(v)w_{ij}\mathbf{P}_{ij}$$

$$w_{,u}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N^p_{i,u}(u)N^q_j(v)w_{ij}$$

A similar surface point derivative could be defined for $\mathbf{S}_{,v}$. These derivatives are useful in the sense of differential geometry of NURBS for several CAD applications [Krishnamurthy et al., 2009]. However, since many deep learning applications such as surface fitting are not dependent on the (u,v) parametric coordinates, we do not use them in our layer. Also, note that $\mathbf{S}_{,u}$ and $\mathbf{S}_{,v}$ are not the same as $\mathbf{S}_{,\mathbf{U}}$ and $\mathbf{S}_{,\mathbf{V}}$. A discussion about $\mathbf{S}_{,\mathbf{U}}$ and $\mathbf{S}_{,\mathbf{V}}$ is provided later in this section. Now, let us define $\mathbf{S}_{,p_{ij}}(u,v)$.

$$\mathbf{S}_{,\mathbf{P}_{ij}}(u,v) = \frac{N^p_i(u)N^q_j(v)w_{ij}}{\sum_{k=0}^{n}\sum_{l=0}^{m} N^p_k(u)N^q_l(v)w_{kl}} \tag{22}$$

$\mathbf{S}_{,\mathbf{P}_{ij}}(u,v)$ is the rational basis functions themselves. Computing $\mathbf{S}_{,w_{ij}}(u,v)$ is more involved with w_{ij} terms in both the numerator and the denominator of the evaluation.

$$\mathbf{S}_{,w_{ij}}(u,v) = \frac{\mathbf{NR}_{,w_{ij}}(u,v)w(u,v) - \mathbf{NR}(u,v)w_{,w_{ij}}(u,v)}{w(u,v)^2} \tag{23}$$

where,

$$\mathbf{NR}_{,w_{ij}}(u,v) = N^p_i(u)N^q_j(v)\mathbf{P}_{ij}$$

$$w_{,w_{ij}}(u,v) = N^p_i(u)N^q_j(v)$$

C.3 Derivatives with respect to knot points

For simplicity, we will stick to 1D NURBS curves. The extension to 2D surfaces is straightforward using Kronecker products.

We recall the definition of the NURBS basis:

$$N^d_i(u) = \frac{u-u_i}{u_{i+d}-u_i}N^{d-1}_i(u) + \frac{u_{i+d+1}-u}{u_{i+d+1}-u_{i+1}}N^{d-1}_{i+1}(u),\ N^0_i(u) = \begin{cases} 1 & \text{if } u_i \le u \le u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

The goal is to evaluate the derivative of $N^d_i(u)$ with respect to the knot points $\{u_i\}$. We observe that due to the recursive nature of the definition, we can accordingly compute the derivatives of $N^d_i(u)$ in a recursive fashion using the chain rule, *provided* we can evaluate:

$$\frac{\partial N^0_i(u)}{\partial u_i} = \frac{\partial \mathbf{1}([u_i, u_{i+1}])}{\partial u_i}$$

(and likewise for u_{i+1}) where $\mathbf{1}$ denotes the indicator function over an interval. However, this derivative is not well-defined since the gradient is zero everywhere and undefined at the interval edges.

We propose to approximate this derivative using *Gaussian smoothing*. Rewrite the interval as the difference between step-functions convolved with deltas shifted by u_i and u_{i+1} respectively:

$$\mathbf{1}([u_i, u_{i+1}))(u) = \text{sign}(u) \star \delta(u - u_i) - \text{sign}(u) \star \delta(u - u_{i+1})$$

and approximate the delta function with a Gaussian of sufficiently small (but constant) bandwidth:

$$\mathbf{1}([u_i, u_{i+1}])(u) = \text{sign}(u) \star G_\sigma(u - u_i) - \text{sign}(u) \star G_\sigma(u - u_{i+1})$$

where

$$G_\sigma(u - \mu) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(u-\mu)^2}{2\sigma^2}).$$

The derivative with respect to μ is therefore given by:

$$G'_\sigma(u = \mu) = \frac{(u-\mu)}{2\sigma^2} G_\sigma(u - \mu),$$

which means that the approximate gradient introduces a multiplicative $(u-\mu)$ factor with the original basis function. Propagating this through the chain rule and applying a similar strategy as Cox-de Boor recursion gives us Algorithm 1. □

D Experimental details

D.1 Segmentation

Weizmann Horse dataset: The dataset consists of 378 images of single horses with varied backgrounds and their corresponding ground truth. We divide the dataset into 85:15 ratios for training and testing, respectively. Further, each image is normalized to a $[0, 1]$ domain by dividing it by 256. 5443

Cell dataset: The dataset consists of 19K gray-scale images containing various cells, and we take 1900 subset images as the dataset. We divide the dataset into 85:15 ratios for training and testing, respectively. Similarly, we normalize the image to a $[0, 1]$ by dividing each pixel by 256.

Architecture and training: We use the following U-Net architecture for training our segmentation networks. While we use the equivalent model skeleton reported by Ronneberger et al. [2015], we scale down the network size starting the initial channels $C = 8$ (default channel is $C = 64$). In both datasets, we train the network 1000 epochs with an initial learning rate of 0.0003. We leverage Adam optimizer with $\beta = (0.9, 0.999)$ and weight decay 0.0001. We use a binary cross-entropy loss function as the objective function.

D.2 NURBS surface fitting implementation

The complete algorithm for forward evaluation of $\mathbf{S}(u, v)$ as described in Piegl and Tiller [1997] can be divided into three steps:

1. Finding the knot span of $u \in [u_i, u_{i+1})$ and the knot span of $v \in [v_j, v_{j+1})$, where $u_i, u_{i+1} \in \mathbf{U}$ and $v_j, v_{j+1} \in \mathbf{V}$. This is required for the efficient computation of only the non-zero basis functions.
2. Now, we compute the non-zero basis functions $N_i^p(u)$ and $N_j^q(v)$ using the knot span. The basis functions have specific mathematical properties that help us in evaluating them efficiently. The partition of unity and the recursion formula ensures that the basis functions are non-zero only over a finite span of $p + 1$ control points. Therefore, we only compute those $p + 1$ non-zero basis functions instead of the entire n basis function. Similarly in the v direction we only compute $q + 1$ basis functions instead of m.
3. We first compute the weighted control points $\mathbf{P}_{ij}^w$ for a given control point $\mathbf{P}_{ij} = \{\mathbf{P}_x, \mathbf{P}_y, \mathbf{P}_z\}$ and weight w_{ij} as $\{\mathbf{P}_x w, \mathbf{P}_y w, \mathbf{P}_z w\}$ representing the surface after homogeneous transformation for ease of computation. Once the basis functions are computed we multiply the non-zero basis functions with the corresponding weighted control points, $\mathbf{P}_{ij}^w$. This result, $\mathbf{S}'$ is then used to compute $\mathbf{S}(u, v)$ as $\{S'_x/S'_w, S'_y/S'_w, S'_z/S'_w\}$.

Algorithm 2 Forward algorithm for multiple surfaces

Input : $\mathbf{U}$, $\mathbf{V}$, $\mathbf{P}$, $\mathbf{W}$, output resolution n_{grid}, m_{grid}
Output : $\mathbf{S}$
Initialize a meshgrid of parametric coordinates
    uniformly from $[0, 1]$ using $n_{grid} \times m_{grid} : u_{grid} \times v_{grid}$
Initialize: $\mathbf{S} \rightarrow \mathbf{0}$
for $k = 1 : surfaces$ *in* **parallel do**
  for $j = 1 : m_{grid}$ *points in* **parallel do**
    for $i = 1 : n_{grid}$ *points in* **parallel do**
      Compute u_{span} and v_{span} for the corresponding u_i and v_i using knot vectors $\mathbf{U_k}$ and $\mathbf{V_k}$
      Compute basis functions N_i and N_j basis functions using u_{span} and v_{span} and knot vectors $\mathbf{U_k}$ and $\mathbf{V_k}$
      Compute surface point $\mathbf{S}(u_i, v_j)$ (in x, y, and z directions).
      Store u_{span}, v_{span}, N_i, N_j, and $\mathbf{S}(u_i, v_j)$ for backward computation

In a deep learning system, each layer is considered as an independent unit performing the computation. The layer takes a batch of input during the forward pass and transforms them using the parameters of the layer. Further, in order to reduce the computations needed during the backward pass, we store extra information for computing the gradients during the forward computation. The NURBS layer takes as input the control points, weights, and knot vectors for a batch of NURBS surfaces. We define a parameter to control the number of points evaluated from the NURBS surface. We define a mesh grid of a uniformly spaced set of parametric coordinates $u_{grid} \times v_{grid}$. We perform a parallel evaluation of each surface point $S(u, v)$ in the $u_{grid} \times v_{grid}$ for all surfaces in the batch and store all the required information for the backward computation. The complete algorithm is explained in Algorithm 2. Our implementation is robust and modular for different applications. For example, if an end-user desires to use this for a B-spline evaluation, they need to set the knot vectors to be uniform and weights $\mathbf{W}$ to be 1.0. In this case, the forward evaluation can be simplified to $\mathbf{S}(u, v) = \mathbf{f}(\mathbf{P})$. Further, we can also pre-compute the knot spans and basis functions during the initialization of the NURBS layer. During computation, we could make use of tensor comprehension that significantly increases the computational speed. We can also handle NUBS (Non-Uniform B-splines), where the knot vectors are still non-uniform, but the weights W are set to 1.0. Note in the case of B-splines $\Psi = \{\mathbf{P}\}$ (the output from the deep learning framework) and in the case of NUBS $\Psi = \{\mathbf{P}, \mathbf{U}, \mathbf{V}\}$.

SplineNet training details: The SplineNet architecture comprises a series of dynamic graph convolution layers, followed by an adaptive max pooling and conv1d layers. We use the Chamfer distance as the loss function. The Chamfer distance ($\mathcal{L}_{CD}$) is a global distance metric between two sets of points, as shown below.

$$\mathcal{L}_{CD} = \sum_{\mathbf{P_i} \in \mathbf{P}} \min_{\mathbf{Q_j} \in \mathbf{Q}} ||\mathbf{P_i} - \mathbf{Q_j}||_2 + \sum_{\mathbf{Q_j} \in \mathbf{Q}} \min_{\mathbf{P_i} \in \mathbf{P}} ||\mathbf{P_i} - \mathbf{Q_j}||_2 \tag{25}$$

For training and testing our experiments, we use the SplineDataset provided by Sharma et al. [2020]. The SplineDataset is a diverse collection of open and closed splines that have been extracted from one million CAD geometries included in the ABC dataset. We run our experiments on open splines split into 3.2K, 3K, and 3K surfaces for training, testing, and validation.

D.3 PDE solver implementation with DSA prior

Deep convolutional neural networks are a natural choice for the network architecture for solving PDEs due to the structured grid representation of $\mathcal{S}^d$ and similarly structured representation of U_θ^d. The spatial localization of convolutional neural networks helps in learning the interaction between the discrete points locally. Since the network takes an input of a discrete grid representation (similar to an image, possibly with multiple channels) and predicts an output of the solution field of a discrete grid representation (similar to an image, possibly with multiple channels), this is considered to be similar to an image segmentation or image-to-image translation task in computer vision. U-Nets [Ronneberger et al., 2015] have been known to be effective for applications such as semantic segmentation and image reconstruction.

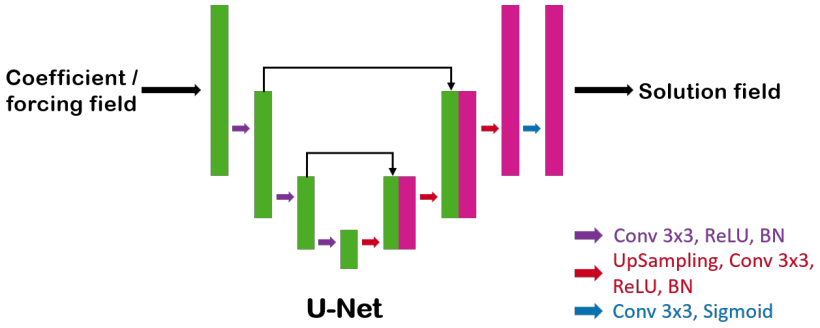

Figure 4: UNet architecture used for training

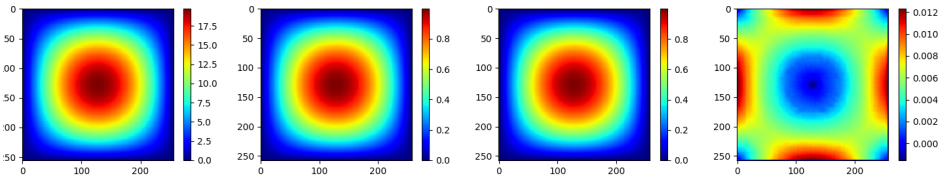

Figure 5: Solution to the linear Poisson's equation with forcing. From left to right: f, u_{DSA}, u_{num} and $(u_{DSA} - u_{num})$. Here u_{num} is a conventional numerical solution obtained through FEM. Diffusivity $\nu = 1$

We choose U-Net architecture for solving the PDE due to its success in other diverse applications. The architecture of the network is shown in Figure 4. First, a block of convolution and instance normalization is applied. Then, the output is saved for later use during skip-connection. This intermediate output is then downsampled to a lower resolution for a subsequent convolution block and instance normalization layers. This process is continued twice. The upsampling starts where the saved outputs of similar dimensions are concatenated with the output of upsampling for creating the skip-connections followed by a convolution layer. LeakyReLU activation was used for all the intermediate layers. The final layer has a Sigmoid activation.

D.3.1 Applying boundary conditions

The Dirichlet boundary conditions are applied exactly. The query result from U_θ^d from the network pertains only to the interior of the domain. The boundary conditions need to be taken into account separately. There are two ways of doing this:

- Applying the boundary conditions exactly (this is possible only for Dirichlet conditions in FEM/FDM, and the zero-Neumann case in FEM)
- Taking the boundary conditions into account in the loss function, thereby applying them approximately.

We take the first approach of applying the Dirichlet conditions exactly (subject to the mesh). Since the network architecture is well suited for 2d and 3d matrices (which serve as an adequate representation of the discrete field in 2D/3D on regular geometry), the imposition of Dirichlet boundary conditions amounts to simply padding the matrix by the appropriate values. A zero-Neumann condition can be imposed by taking the "edge values" of the interior and copying them as padding. A nonzero Neumann condition is slightly more involved in the FDM case since additional equations need to be constructed, but if using FEM loss, this can be done with another surface integration on the relevant boundary.

E Additional results

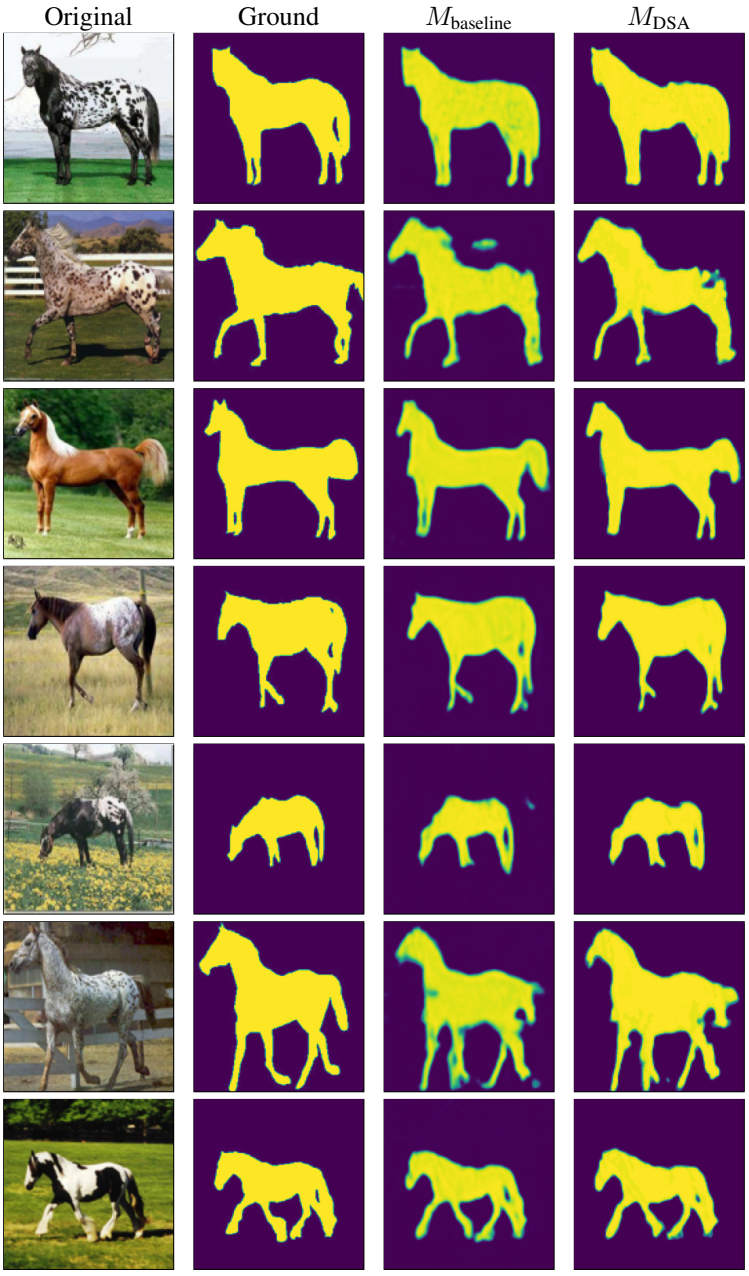

Figure 6: **Image Segmentation Tasks** Adding DSA layers (M_{DSA}) on top of U-Net (M_{baseline}) improves the segmentation tasks on both datasets.

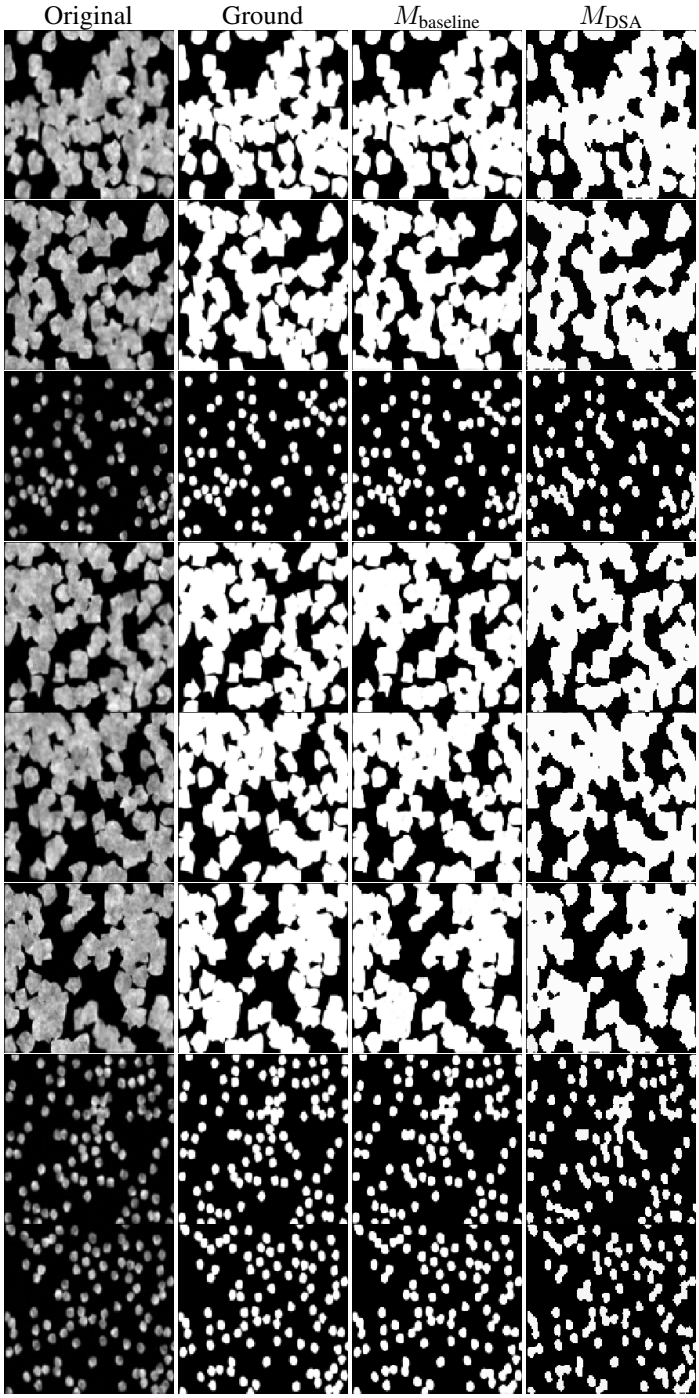

Figure 7: Additional cell segmentations results. M_{baseline} and M_{DSA} correspond to U-Net and U-Net+DSA layers, respectively.

Ablation studies: In the main paper, we demonstrate how our module can perform for different experiments. In this section, we assess the computational performance of our module. For brevity, we restrict our analysis to surface fitting operation and analyze the timings with variations in the number of control points, evaluation points, and surface degree. We only study the first 500 iterations (which include both the forward and backward pass). We perform all our experiments on a desktop with a 32 core 2.4 GHz Intel Xeon processor, 64 GB RAM, and an NVIDIA Titan Black GPU with 6 GB RAM.

Table 6: Time to fit a surface for different number of control points.

Control Points	**Iteration time (s)**
6×6	0.098
12×12	0.106
24×24	0.110
48×48	0.110

Table 7: Time to fit a surface for different number of evaluation points.

Evaluation Points	**Iteration time (s)**
64×64	0.074
128×128	0.120
256×256	0.170
512×512	0.266

Table 8: Computation time to fit a surface of different degree.

Degree	**Iteration time (s)**
1	0.074
2	0.120
3	0.170
4	0.266