

Polynomial-Spline Neural Networks with Exact Integrals

Jonas A. Actor,¹ Andy Huang,² Nathaniel Trask¹

Affiliations ¹Center for Computing Research, Sandia National Laboratories

²Radiation and Electrical Science, Sandia National Laboratories

1515 Eubank Blvd SE

Albuquerque, NM 87123

jaactor@sandia.gov, ahuang@sandia.gov, natrask@sandia.gov

Abstract

Using neural networks to solve variational problems, and other scientific machine learning tasks, has been limited by a lack of consistency and an inability to exactly integrate expressions involving neural network architectures. We address these limitations by formulating a novel neural network architecture incorporating free knot B1-spline basis functions into a polynomial mixture-of-experts model. Effectively, our architecture performs piecewise polynomial approximation on each cell of a trainable partition of unity while ensuring the neural network and its derivatives can be integrated exactly, obviating a reliance on sampling or quadrature and enabling error-free computation of variational forms. We demonstrate hp -convergence for regression problems at convergence rates expected from approximation theory and solve elliptic problems in one and two dimensions to show an effective dimension reduction compared to adaptive finite elements.

Introduction

Deep neural networks (DNNs) have been proposed for solving partial differential equations (PDEs) and scientific machine learning, but fail to converge to PDE solutions in practical settings (Beck, Jentzen, and Kuckuck 2020; Han, Jentzen, and Weinan 2018). Notably, partition of unity networks (POUNets) (Lee et al. 2021) demonstrate hp -convergence for regression problems, in terms of spatial resolution h and polynomial degree p , similar to finite element methods. However, use of inexact quadrature is a *variational crime* (Strang 1972) complicating error analysis for these and other architectures; such issues have led to the popularity of simpler to implement but more difficult to analyze collocation schemes (Raissi, Perdikaris, and Karniadakis 2019; Wang, Teng, and Perdikaris 2020).

In this work, we introduce polynomial-spline neural networks, a mixture-of-experts (MOE) model that combines gating functions composed of convex combinations of B-spline basis functions, with polynomial experts localized to each cell of the partition. The spline gating functions lead to a piecewise polynomial approximation with explicitly parameterized support, creating closed-form expressions for the integral of the DNN and its derivatives. As such, this

provides a foundation for other problems requiring integration, e.g. estimation of statistical moments for probability measures, or novel loss functions and regularizers.

Related Work

Our architecture admits interpretation as a MOE model (Yuksel, Wilson, and Gader 2012) with gating functions provided by a convex combination of B-spline basis functions rather than a hidden layer with softmax activation. While similar to MOE, POUNets (Lee et al. 2021) aim to build hp -convergent solutions to regression/PDE problems rather than perform kernel density estimation. Unlike the multi-layer perceptron (MLP) or radial basis function network considered in (Lee et al. 2021), working with convex combinations of B-splines admits closed form expressions for integrals in terms of the B-spline knots. We thus preserve the hp -convergence of POUNets while enabling exact integration, even in high-dimensional settings; Compared to traditional B-spline bases the convex combination identifies an optimal compressed subspace for nonlinear approximation.

Our B-splines provide natural extensions of well-known results interpreting ReLU networks as continuous piecewise linear (CPWL) functions (Arora et al. 2018). Previous works use splines to study approximation properties of DNNs, treating e.g. ReLU networks as max-affine splines (Balestrieri and Baraniuk 2020) or as P1 finite elements (He et al. 2020). In high-dimensions, the CPWL interpretation of ReLU networks is not tractable for quadrature, as the geometrically complex piecewise linear regions form non-convex polyhedral domains and do not admit a closed-form description of their support. In light of these results, training a ReLU DNN is similar in scope to finding an optimal spline interpolant with trainable knots (Balestrieri and Baraniuk 2020), an NP-hard problem (Jupp 1978). In the related problem of adaptively finding an optimal mesh for discretizing PDEs, the mesh adaptivity is guided by an energy functional relating to the discretization points, whose minimizer is the optimal adaptive mesh. Such an energy function again necessitates integrating over the variational form of the PDE (Babuvska and Rheinboldt 1978; Gui and Babuska 1986; Logg et al. 2012), which precludes their use in training DNNs.

Some PDE discretizations of DNNs skirt the issue of inte-

gration by adapting a collocation PDE residual (e.g. physics-informed neural networks (PINNs) (Raissi, Perdikaris, and Karniadakis 2019) and related methods). While effective, this requires strong regularity requirements and more involved mathematical analysis beyond the standard Lax-Milgram theory (Shin 2020). Alternatively, the Deep Ritz method uses as a loss the Euler-Lagrange functional of the relevant variational problem, but ultimately resorts to sampling-based methods for integration (E and Yu 2018). As a result, the convergence of the loss function is dominated by the error in Monte Carlo integration, and variational crimes complicate the already complex landscape of approximation error, optimization error, and stability theory. An important practical feature of preserving the variational form is that we may train by evolving along the manifold of optimal fits to data, similar to the least-squares gradient descent optimizer (Cyr et al. 2020).

Formulation

Let $\Omega \subset \mathbb{R}^d$ be a closed, compact domain, where d is the spatial dimension; assume for simplicity that $\Omega = [0, 1]^d$. Let $\mathbb{P}^B(\Omega)$ be the space of polynomials of degree up to B on Ω , with basis $\{p_\beta\}_{\beta=1,\dots,d_P=\dim(\mathbb{P}^B(\Omega))}$.

We define a *polynomial-spline neural network* $y : \Omega \rightarrow \mathbb{R}$ via the expression

$$y(x) = \sum_{\alpha=1}^{N_{\text{cells}}} \left(\sum_{\gamma=0}^{N_{\text{splines}}} w_{\alpha,\gamma} \phi_\gamma(x) \right) \left(\sum_{\beta=1}^{d_P} c_{\alpha,\beta} p_\beta(x) \right). \quad (1)$$

In this expression, the functions $\phi_\gamma : \Omega \rightarrow \mathbb{R}$ are B-spline basis functions, parameterized by a set of knots $\{t_\gamma\} \subset \Omega$. Additionally the coefficients $w_{\alpha,\gamma}$ are constrained so that for all γ , $\sum_{\alpha=1}^{N_{\text{splines}}} w_{\alpha,\gamma} = 1$ and $w_{\alpha,\gamma} \geq 0$. When clear, the bounds for the summations in Equation (1) are dropped for convenience.

The functions $\varphi_\alpha(x) = \sum_{\gamma=1}^{N_{\text{splines}}} w_{\alpha,\gamma} \phi_\gamma(x)$ form a partition of unity (POU) of N_{cells} partitions, following from convexity of $W_{\alpha,\gamma}$ and the fact that B-spline basis functions form a partition of unity (Powell et al. 1981). Thus, polynomial-spline networks are MOE models, where convex combinations of B-splines serve as gating functions for B^{th} -order polynomial experts. In the case that $N_{\text{cells}} = 1$, training this architecture reduces to polynomial approximation, as the polynomial-spline network becomes

$$\begin{aligned} y(x) &= 1 \cdot \left(\sum_{\beta} c_{\beta} p_{\beta}(x) \right) \\ &= \sum_{\beta} c_{\beta} p_{\beta}(x). \end{aligned}$$

Similarly, in the case $B = 0$ and $N_{\text{cells}} = N_{\text{splines}}$, training this architecture reduces to free-knot spline approximation, since the network becomes

$$\begin{aligned} y(x) &= \sum_{\alpha} \left(\sum_{\gamma} w_{\alpha,\gamma} \phi_\gamma(x) \right) (c_{\alpha}) \\ &= \sum_{\alpha} \sum_{\gamma} c_{\alpha} w_{\alpha,\gamma} \phi_\gamma(x), \end{aligned}$$

and setting $w_{\alpha,\gamma} = \delta_{\alpha\gamma}$ reduces the architecture to

$$y(x) = \sum_{\alpha} c_{\alpha} \phi_{\alpha}(x).$$

Therefore, we expect our polynomial-spline network to exhibit some form of both h - and p -refinement, as the number of partitions and polynomial degree increase, respectively, following the convergence rates established via numerical analysis e.g. (Suli and Mayers 2003).

In practice, we limit ourselves to only using B1-splines. In doing so, we make the max-affine spline interpretation of deep ReLU networks (Balestrieri and Baraniuk 2020) explicit, in that we directly construct the underlying spline to partition Ω . Doing so allows us to construct closed-form expressions for the integrals (and integrals of derivatives) of the polynomial-spline network; deriving such expressions is tedious but feasible for higher-order splines.

We outline how to construct analytic expressions for the integral of y in the case of the functions ϕ_γ being B1-spline basis functions and our domain $\Omega = [0, 1]$. First, note that the B1-spline basis functions are described entirely by the set of knots $\{t_\gamma\}_{\gamma=0,\dots,N_{\text{splines}}}$, with $t_0 = 0$ and $t_{N_{\text{splines}}} = 1$, with relation to ϕ_γ in that $\phi_\gamma(t_\gamma) = 1$ and $\phi_\gamma(t_\beta) = 0$ for all $\beta \neq \gamma$. By construction, when restricted to the interval $[t_{\gamma-1}, t_\gamma]$, the polynomial-spline network y is a polynomial of degree $B + 1$. Letting $\{q_i\}_{i=1,\dots,d_P+1}$ be a basis for $\mathbb{P}^{B+1}([t_{\gamma-1}, t_\gamma])$, we express y restricted to our interval in this basis, i.e.

$$\begin{aligned} y(x) &= \sum_{\alpha} (w_{\alpha,\gamma-1} \phi_{\gamma-1}(x) + w_{\alpha,\gamma} \phi_{\gamma}(x)) \left(\sum_{\beta} c_{\alpha\beta} p_{\beta}(x) \right) \\ &:= \sum_{i=1}^{d_P+1} d_i q_i(x) \end{aligned}$$

for coefficients d_i , which are closed-form expressions of the coefficients $w_{\alpha,\gamma}$, t_γ , and $c_{\alpha\beta}$. Therefore, our integral becomes

$$\begin{aligned} \int_{\Omega} y(x) dx &= \sum_{\gamma=1}^{N_{\text{splines}}} \int_{t_{\gamma-1}}^{t_\gamma} y(x) dx \\ &= \sum_{\gamma=1}^{N_{\text{splines}}} \sum_{i=1}^{d_P+1} d_i \int_{t_{\gamma-1}}^{t_\gamma} q_i(x) dx. \end{aligned}$$

Choosing the monomial basis $q_i(x) = x^{i-1}$,

$$\begin{aligned} \int_{\Omega} y(x) dx &= \sum_{\gamma=1}^{N_{\text{splines}}} \sum_{i=1}^{d_P+1} d_i \int_{t_{\gamma-1}}^{t_\gamma} x^{i-1} dx \\ &= \sum_{\gamma=1}^{N_{\text{splines}}} \sum_{i=1}^{d_P+1} \frac{d_i}{i} (t_\gamma^i - t_{\gamma-1}^i). \end{aligned}$$

Since we can explicitly calculate expressions for d_i , and all other values are known weights in our network, we can use the above formula to directly integrate y . Similar expressions are derived in the same way for ∇y , or for the calculation of moments involving y to a power, by expressing the integrand in terms of a polynomial expansion in terms of the polynomials q_i and then integrating separately upon the support of each B1-spline basis function.

Experiments

We demonstrate the effectiveness of our architecture on two sets of problems: regression problems and variational problems. Our implementation of polynomial-spline neural networks, and all the related layers and training strategies described herein, are implemented using TensorFlow (Abadi et al. 2015); we use NumPy (Harris et al. 2020), SciPy (Virtanen et al. 2020), and FEniCS (Alnæs et al. 2015) to compare our results to classical methods.

Construction

To build our polynomial-spline network, we build B1-spline basis functions for Ω as a tensor product of 1D B1-spline basis functions along each dimension. For each dimension, we construct a B1-spline layer, whose knots are parameterized to accommodate TensorFlow backwards differentiation during training. The general expression for a hat function built via ReLU functions is given in (He et al. 2020). During training however, knots may become unordered, leading to a problematic inversion of elements. To prevent this concern, we track the relative position between knots rather than the locations themselves, constraining them to span the extant Ω . For more details, please see the supplementary material.

Expedited Training via LSGD

We expedite the training of our models by using the least-squares gradient descent optimizer(LSGD) (Cyr et al. 2020). We define the function $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^{N_{\text{cells}} d_P}$ as $\Phi_{\alpha\beta} : x \mapsto \left(\sum_{\gamma} w_{\alpha,\gamma} \phi_{\gamma}(x) \right) p_{\beta}(x)$, and rewrite Equation 1 as

$$y(x) = c^T \Phi(x)$$

for a vector of coefficients $c \in \mathbb{R}^{N_{\text{cells}} d_P}$. For regression problems, the LSGD solver adds a least-squares solve for c between each gradient step of the first-order optimizer. We wrap this least-squares solve call into a custom TensorFlow layer so that the operation is embedded in the network’s TensorFlow graph directly, enforcing that all outputs of the network lie on the manifold of best-fit solutions regarding the coefficients of the outermost layer. We note that there are stability issues with the implementation of the backwards differentiation of the least-squares function call in TensorFlow, resulting in accuracies at most of order $O(10^{-8})$ instead of machine-epsilon precision. Other implementations of this method do not suffer from this issue and can achieve errors as low as $O(10^{-20})$ and beyond; see the supplementary material for more details.

For regression, LSGD solves the least-squares problem, given batch data $\{x_i, y_i\}_{i=1, \dots, \text{batch size}}$ the problem

$$\min_c \|y - c^T \Phi(x)\|_F^2.$$

For variational problems, the least-squares problem that we solve is the variational problem that corresponds to the Euler-Lagrange functional; see the discussion below about our variational problems for more details.

Training Details

For the regression problems, we train on a random uniform set of 1000 points, and we validate our model on a separate random uniform set of the same size. For the variational problems, there are no data sites involved, with the loss defined via the closed form expression for the energy. For all problems we use the Adam optimizer (Kingma and Ba 2014). Our loss for the regression problems is mean squared error (MSE), while for the variational problems our loss is the Euler-Lagrange functional i.e. the Ritz energy. More details, including a description of hyperparameters, are available in the supplementary material. Code for each problem is available in the supplementary material. All code is run on a 64-core Intel Xeon Gold CPU running Linux with 10 NVIDIA Tesla V100 GPUs with 32GB each, although code was restricted at runtime to only use 1 GPU and 10GB of memory for portability reasons.

Problems

To demonstrate the effectiveness of our architecture, we pose two sets of problems. First, we deploy our architecture on regression problems to evaluate consistency. After, we progress to solving variational problems.

Regression We test our architecture on two 1D regression problems:

1. $f(x) = \sin(2\pi x)$ for $x \in [0, 1]$
2. $f(x) = |\sin(3\pi x^2)| + |\cos(5\pi x^2)|$ for $x \in [0, 1]$

In Problem 1, we expect to see both $h-$ and $p-$ refinement, i.e. increasing the number of partitions or increasing the degree of polynomial approximation, respectively, should improve the approximation. In Problem 2, we expect $h-$ refinement to improve our approximation but $p-$ refinement to not, since the function f in this case is only piecewise smooth. However, if we have a sufficient number of knots (i.e. at least one per piece of f), our POU cells should adaptively during training recover the locations where the sin or cos terms change sign, and that on each piece, we expect $p-$ refinement to improve our approximation.

Problem 1. We perform three experiments to demonstrate the $hp-$ refinement properties of our network. First, we fix $N_{\text{cells}} = 1$ to verify that our model compares favorably to polynomial approximation with regards to $p-$ refinement. Second, we fix the polynomial degree $B = 0$ to verify that our model compares favorably to spline approximation with regards to $h-$ refinement. Third, we test our model for a variety of parameters for polynomial degree, number of cells, and number of spline knots, to verify simultaneous $hp-$ refinement and to test that our model can capture the solution to this regression problem up to machine precision.

First, we consider Problem 1 with $N_{\text{cells}} = 1$; in this case, we expect our model to return the best polynomial approximation of the specified degree. Results are shown in Figure 1; all three lines plotted in the figure nearly coincide, showing that in this limit our model maintains the $p-$ refinement properties of polynomial approximation. The staircase phenomenon is due to the function $f(x) = \sin(2\pi x)$ being an

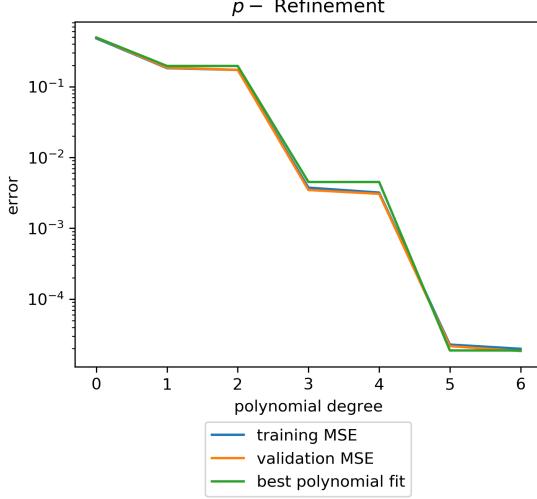


Figure 1: Results of $p-$ refinement for regression Problem 1, with $N_{\text{spline}} = 4$ and $N_{\text{cells}} = 1$. The errors for the training and validation sets coincide with the error for the best polynomial approximation.

odd function, and as such the best polynomial approximation (and our model) only improves when the polynomial degree is increased to an odd power.

Second, to verify the $h-$ refinement properties of our architecture, we set the polynomial degree $B = 0$ and compare our results to a piecewise linear spline approximation with uniform knots. Error plots are shown in Figure 2. In this case, we see the roughly the same rate at which the error decreases as we increase the number of knots in the network and in our spline approximation, until our model plateaus due to the instability of the backwards differentiation in the LSGD layer.

Finally, when using $h-$ and $p-$ refinement simultaneously the polynomial-spline network is capable of achieving machine precision accuracy. Results are shown in Figure 3. We see that for sufficient spatial resolution and polynomial degree, we achieve mean-squared errors of order 10^{-8} or better using our proposed network, which is the precision limit given by our least-squares implementation in the LSGD layer.

Problem 2. We perform two sets of experiments for Problem 2. First, we perform the same experiment as for Problem 1 to test $h-$ refinement even in the case of a non-smooth target function. Second, we test whether our network can find the discontinuities in the derivative of f in a way that is comparable to piecewise polynomial approximation. As in the previous problem, the training and validation error curves are extremely similar, so we only plot validation errors in the rest of the figures in this section.

First, as before, we compare our network's performance when the polynomial degree $B = 0$ and compare to a spline approximation with the same number of knots. In Figure

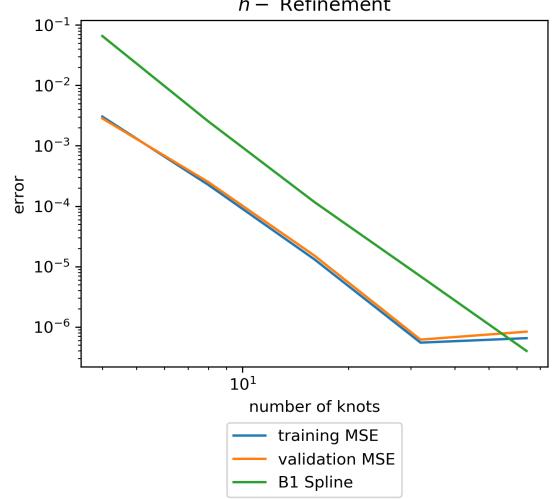


Figure 2: Results of $h-$ refinement for regression Problem 1, with $N_{\text{cells}} = N_{\text{spline}}$ and polynomial degree $B = 0$. The x -axis maintains a logarithmic scale.

4, we see the error in our approximations for increasingly larger numbers of knots; we observe a roughly log-linear decrease in mean squared error as we double the number of knots (and POU basis functions) in our network, as expected.

Second, we compare the results of our model with comparable piecewise polynomial approximation problems. We specifically compare to two piecewise polynomial approximations, one where polynomials are fit upon uniform pieces, and the other fit to pieces where the derivative of f is discontinuous. As both of these piecewise polynomial approximants fit a total of 8 polynomials, we compare these results to our model with $N_{\text{cells}} = 8$, which fits 8 polynomials in the LSGD layer. Results are shown in Figure 5. Our polynomial-spline network outperforms these model for all polynomial degrees, efficiently capturing the discontinuous derivatives in the target function. The success of the polynomial-spline network plateaus at an accuracy of $O(10^{-6})$ due to the limitations of the numerical stability in the automatic differentiation of the least-squares solve operation in the LSGD layer.

Variational Problems We test our architecture on two variational problems:

3. 1D Poisson problem with Dirichlet boundary conditions:

$$\begin{aligned} -d^2 u &= 2 & \text{on } \Omega &= [0, 1] \\ u &= 0 & \text{at } \partial\Omega &= \{0, 1\} \end{aligned} \quad (2)$$

4. 2D Poisson problem on a slit domain:

$$\begin{aligned} -\Delta u &= 0 & \text{on } \Omega &= [-1, 1]^2 \\ u &= g(r, \theta) & \text{on } \Gamma &= \partial\Omega \cup [0, 1] \times \{0\}, \end{aligned} \quad (3)$$

where $g(r, \theta) = \sqrt{r} \sin(\frac{\theta}{2})$ is given in polar coordinates.

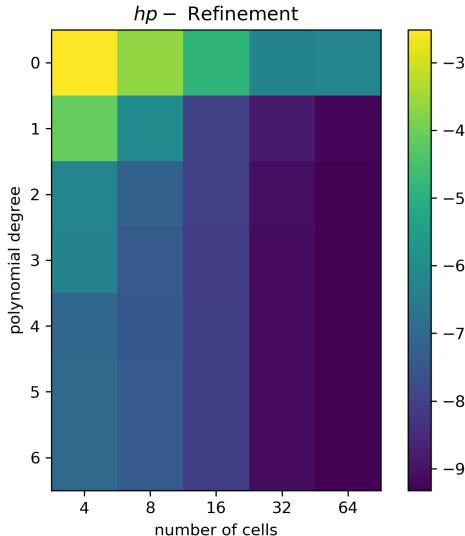


Figure 3: Results for $hp-$ refinement on regression Problem 1. The color bar maps the \log_{10} of the mean squared error for the specified number of POU cells and polynomial degree.

In Problem 3, we expect to recover the true solution $u(x) = x(1 - x)$ exactly (up to machine precision), assuming on each POU cell we are fitting a polynomial of sufficient degree. In Problem 4 we expect our problem to recover the known singularity at the origin similar to adaptive mesh refinement methods (Babuvska and Rheinboldt 1978; Gui and Babuska 1986; Logg et al. 2012); the true solution to this problem is known to be $u = g(r, \theta)$.

For each of these variational problems, we minimize the associated Euler-Lagrange functional (Evans 2010). For example, Problem 3 is solved by minimizing the Euler-Lagrange functional “loss”

$$L(u) = \int_{\Omega} \frac{1}{2} \|\nabla u\|^2 - 2u + \beta (u(0)^2 + u(1)^2), \quad (4)$$

where $\beta > 0$ is a penalty parameter to enforce our solutions satisfy our boundary conditions. For Problem 4, the Euler-Lagrange functional is

$$L(u) = \int_{\Omega} \frac{1}{2} \|\nabla u\|^2 + \beta \int_{\Gamma} (u - g(r, \theta))^2. \quad (5)$$

Instead of relying on Monte Carlo or sampling-based methods to evaluate these integral (e.g. (E and Yu 2018)), we compute these integrals exactly by virtue of our architecture construction. Rather than using the analytic expressions for the integral, we employ Gaussian quadrature of degree $B + d + 1$, which computes the integrals exactly for polynomials of degree up to $2(B + d) + 1$, which is sufficient for exactly computing the integrals of u , u^2 , and $\|\nabla u\|^2$ on the support of each B1 basis function; using quadrature allows us to exploit hardware acceleration during computation, since the model evaluation at the quadrature points can more efficiently use the GPUs. We note that since the lo-

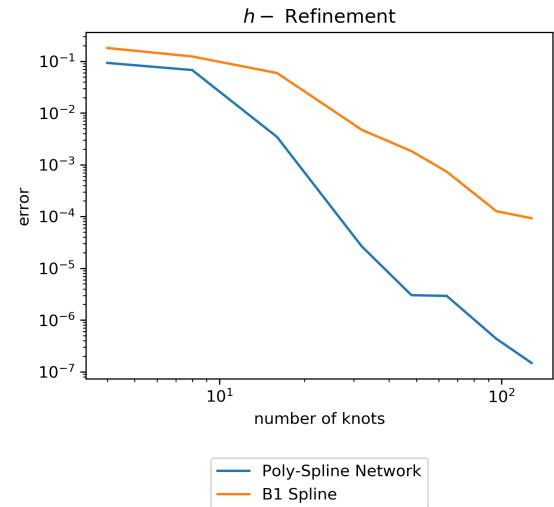


Figure 4: Training results for Problem 2 with $B = 0$ i.e. spline approximation, with $N_{\text{spline}} = N_{\text{cells}}$ plotted against the achieved mean squared error.

cation of the knots change over the course of training, the quadrature points change as well.

Problem 3: 1D Poisson Boundary Value Problem For this problem, the LSGD layer solves the linear system corresponding to the weak form of Problem 3, which is given as follows. We define the matrix A and vector b by substituting $u(x) = c^T \Phi(x)$ into Equation 4, taking the derivative of Equation 4 with respect to c , and then setting the resulting expression to zero. In doing so, we arrive at the linear problem $Ac = b$, where

$$A = \int_{\Omega} d\Phi d\Phi^T + \beta (\Phi(0)\Phi(0)^T + \Phi(1)\Phi(1)^T)$$

$$b = \int_{\Omega} 2\Phi.$$

The integrals in the linear problem can be calculated exactly for a given set of knots, since upon each component of the B1-spline functions, $\Phi(x)$ is a polynomial of degree $B + d$ i.e. $B + 1$. We can do so by deriving the closed-form solutions for the integrands of each interval between spline knots, or (more efficiently) by using Gaussian quadrature of degree $B + d + 1$ as specified earlier.

When we use the training methods described earlier, we successfully solve our problem up to machine precision (or the limits of the accuracy of the least-squares backwards differentiability) nearly immediately when $B \geq 1$, which matches our expectation, since the true solution to our problem is a polynomial of degree 2. Please refer to the supplementary material for more details on these results. In the case $B = 0$, we recover the best P1-finite element solution to our problem, as seen in Figure 6, along with the expected effects of mesh refinement, as seen by the difference in the error plots between the two figures.

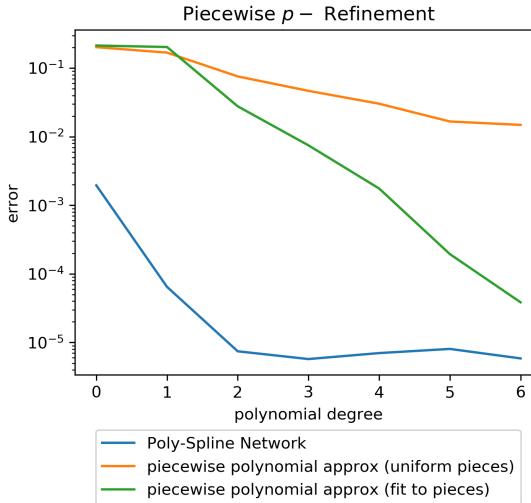


Figure 5: Results of piecewise $p-$ refinement for Problem 2, with $N_{\text{cells}} = 8$.

Problem 4: 2D Poisson Slit Domain By symmetry, it suffices to solve this problem on a reduced domain Ω' that is half the size of Ω : note that the solution to Problem 4 is symmetric about the x-axis. As a result, we solve the following equivalent problem:

$$\begin{aligned} -\Delta u &= 0 && \text{on } \Omega' = [-1, 1] \times [0, 1] \\ \partial_n u &= 0 && \text{on } \Gamma_N = [-1, 0] \times \{0\} \\ u &= g(r, \theta) && \text{on } \Gamma_D = \partial\Omega' \setminus \Gamma_N. \end{aligned} \quad (6)$$

This formulation ensures that the slit in the domain aligns with the exterior of Ω' .

For this problem, the LSGD layer solves the linear system corresponding to the weak formulation of Problem 4, which by the same process as described for Problem 3, yields the linear problem of solving $Ac = b$, where

$$A = \int_{\Omega'} D\Phi D\Phi^T + \beta \int_{\Gamma_D} \Phi \Phi^T, \quad b = \beta \int_{\Gamma_D} g \Phi.$$

Note that the matrix A can be significantly smaller than the linear system involved in other scientific computing methods e.g. the finite element method. For P1 finite elements, the linear system would be of size $d_P (N_{\text{splines}})^d \times d_P (N_{\text{splines}})^d$, whereas in our case A is only of size $d_P N_{\text{cells}} \times d_P \times N_{\text{cells}}$. This reduction in size (and in the cost to solve such problems) arises since we fit polynomials on each partition and not upon each B1-spline basis function. This highlights that our approach provides a nonlinear construction of a reduced finite element space providing an optimal representation of the solution.

The integrals corresponding to A can be calculated via closed-form expressions, as before. However, since g is not a polynomial, we cannot expect exact integration for the integral that defines the vector b , and we either can project g into the correct polynomial space, or over-integrate with

quadrature of a higher degree to maintain sufficient accuracy.

We compare our solution to results of solving this variational problem using P1 finite elements on three different meshes: two uniform tetrahedral meshes, and an adaptive tetrahedral mesh. The first uniform mesh (FEM U3) is a 3×3 mesh, chosen so that there are 16 degrees of freedom and 18 elements, which most closely matches our polynomial-spline construction by fitting linear functions on $N_{\text{cells}} = 16$ cells. The second uniform mesh (FEM U6) is 6×6 , which is the closest match to the size of the linear system that our LSGD layer solves. The adaptive mesh (FEM A) is constructed by an adaptive solver, starting with the 3×3 uniform mesh and proceeding to adaptively refine the mesh until the number of degrees of freedom is closest to the size of our linear system. Near the singularity at $r = 0$, the optimal adaptive mesh's cells should shrink at a rate of approximately \sqrt{r} , where r is their distance to the origin (Logg et al. 2012). Finite element comparisons are computed using FEniCS (Alnæs et al. 2015); for more information on how these comparisons are computed, please refer to the supplementary material.

The results of our method are shown in Figure 7, with the L_2 error listed in Table 1. We outperform the finite element solver on all of the comparable meshes. The plots in Figure 7 re-scale and translate our domain of interest Ω' from $[-1, 1] \times [0, 1]$ to $[0, 1]^2$, with the slit occurring along the line segment $[0.5, 1] \times \{0\}$ along the x -axis.

The errors that arise in our solution accumulate near the boundary, whereas in the finite element approximations, the errors lie in the interior and near the singularity. This is because our Ritz loss enforcing the boundary condition via penalty, while the finite element method enforces Dirichlet conditions variationally. Still, L_2 error of our solution is lower than comparable finite element method solutions, though one could obtain better results working with a boundary conforming Galerkin framework.

Discussion

Our polynomial-spline network compares favorably to traditional approximation methods, preserving the convergence rates of spline interpolation and polynomial regression. Our ability to learn the spline knots and perform free-knot spline interpolation, like other deep learning methods, allows for adaptivity similar to adaptive mesh refinement methods. However, we obtain a reduced partition of space from the overparameterized fine B-spline grid allowing one to work with polynomials on each adaptively coarsened partition of unity; in this sense we obtain reduced-order model for the adaptive B-spline basis. As a result, the size of the linear system involved in the LSGD optimization step is smaller than if we were to depend on the spline basis functions directly. This reduction is particularly effective when $d > 1$, to avoid the curse of dimensionality. Additionally, the number of parameters in our model does not depend on the degree of the polynomial approximation, since the coefficient values for $c_{\alpha\beta}$ are tabulated via LSGD. We thus obtain an hp-convergent variational framework for solving PDEs that circumvents variational crimes due to inexact quadrature.

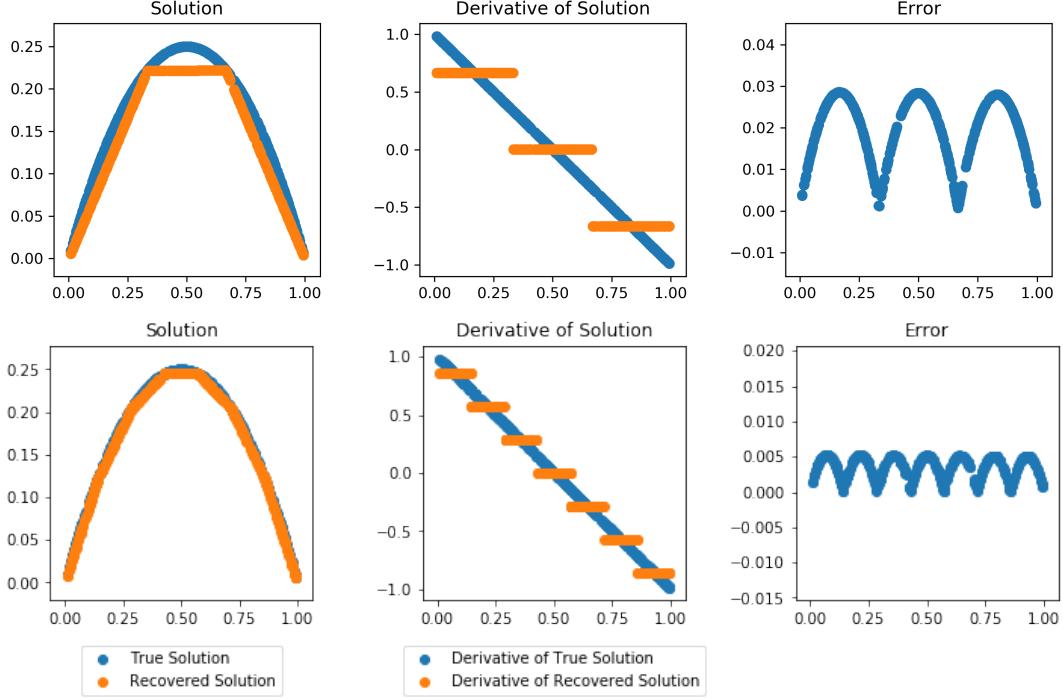


Figure 6: Recovered solution to Problem 3 with $N_{\text{cells}} = 3$, $N_{\text{spline}} = 4$, and $B = 0$ (top) and with $N_{\text{cells}} = 3$, $N_{\text{spline}} = 8$, and $B = 0$ (bottom). We recover the best FEM approximation on a mesh with N_{spline} nodes i.e. $N_{\text{spline}} - 1$ intervals, as seen in the plot of the derivative of the approximation in the center panel.

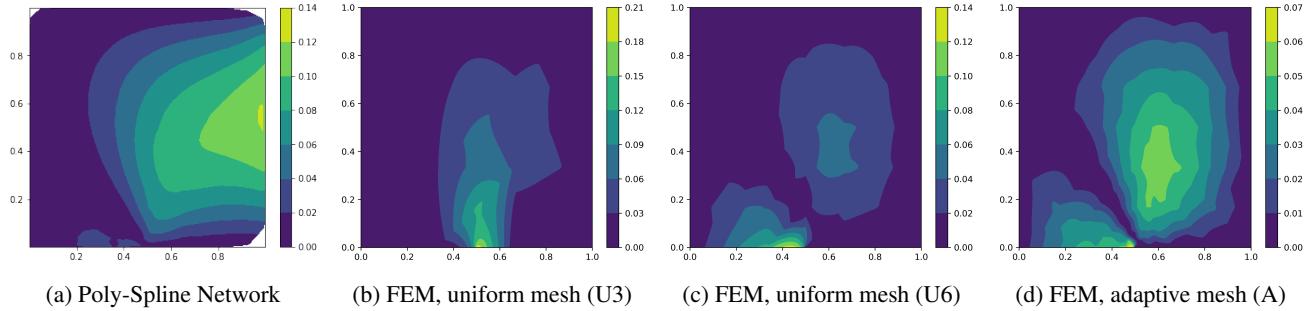


Figure 7: Pointwise error of our model vs. finite element approximations on various meshes. Note the difference in scale.

Method	Mesh Type	# Cells	Solve Size	L_2 Error
Poly-Spline Network	Adaptive	16	48×48	0.0262
FEM U3	Uniform	18	16×16	0.0362
FEM U6	Uniform	72	49×49	0.0231
FEM A	Adaptive	120	72×72	0.0242

Table 1: Comparison of our network vs. adaptive and uniform finite element solutions for Problem 4.

Acknowledgments

N. Trask and J. Actor acknowledge funding under the DOE ASCR PhILMS center (Grant number DE-SC001924) and the DOE Early Career program. A. Huang acknowledges function under the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia

National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

- Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talaraw, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.
- Alnæs, M. S.; Blechta, J.; Hake, J.; Johansson, A.; Kehlet, B.; Logg, A.; Richardson, C.; Ring, J.; Rognes, M. E.; and Wells, G. N. 2015. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3(100).
- Arora, R.; Basu, A.; Mianjy, P.; and Mukherjee, A. 2018. Understanding Deep Neural Networks with Rectified Linear Units. In *International Conference on Learning Representations*.
- Babuvska, I.; and Rheinboldt, W. C. 1978. Error estimates for adaptive finite element computations. *SIAM Journal on Numerical Analysis*, 15(4): 736–754.
- Balestrieri, R.; and Baraniuk, R. G. 2020. Mad max: Affine spline insights into deep learning. *Proceedings of the IEEE*, 109(5): 704–727.
- Beck, C.; Jentzen, A.; and Kuckuck, B. 2020. Full error analysis for the training of deep neural networks. arXiv:1910.00121.
- Cyr, E. C.; Gulian, M. A.; Patel, R. G.; Perego, M.; and Trask, N. A. 2020. Robust Training and Initialization of Deep Neural Networks: An adaptive basis viewpoint. In *Mathematical and Scientific Machine Learning*, 512–536. PMLR.
- E, W.; and Yu, B. 2018. The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems. *Communications in Mathematics and Statistics*, 6(1): 1–12.
- Evans, L. C. 2010. *Partial differential equations*. Graduate studies in mathematics ; Volume 19. Providence, R.I: American Mathematical Society, second edition. edition. ISBN 9780821849743.
- Gui, W.; and Babuska, I. 1986. The h, p and hp versions of the finite element methods in 1 dimension. Part III. The adaptive hp version. *Numerische Mathematik*, 49(6): 659–683.
- Han, J.; Jentzen, A.; and Weinan, E. 2018. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34): 8505–8510.
- Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; del Río, J. F.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; and Oliphant, T. E. 2020. Array programming with NumPy. *Nature*, 585(7825): 357–362.
- He, J.; Li, L.; Xu, J.; and Zheng, C. 2020. ReLU Deep Neural Networks and Linear Finite Elements. *Journal of Computational Mathematics*, 38(3): 502–527.
- Jupp, D. L. 1978. Approximation to Data by Splines with Free Knots. *SIAM Journal on Numerical Analysis*, 15(2): 328–343.
- Kingma, D. P.; and Ba, J. 2014. Adam: A Method for Stochastic Optimization. arXiv:1412.6980.
- Lee, K.; Trask, N. A.; Patel, R. G.; Gulian, M. A.; and Cyr, E. C. 2021. Partition of Unity Networks: Deep HP-Approximation. arXiv:2101.11256.
- Logg, A.; Mardal, K.-A.; Wells, G. N.; et al. 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Springer. ISBN 978-3-642-23098-1.
- Powell, M. J. D.; et al. 1981. *Approximation theory and methods*. Cambridge university press.
- Raissi, M.; Perdikaris, P.; and Karniadakis, G. E. 2019. 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.
- Shin, Y. 2020. On the Convergence of Physics Informed Neural Networks for Linear Second-Order Elliptic and Parabolic Type PDEs. *Communications in Computational Physics*, 28(5): 2042–2074.
- Strang, G. 1972. Variational Crimes in the Finite Element Method. In *The Mathematical Foundations of the Finite Element Method with Applications to Partial Differential Equations*, 689–710. Elsevier.
- Suli, E.; and Mayers, D. 2003. *An introduction to numerical analysis*. Cambridge ;: Cambridge University Press. ISBN 0521810264.
- Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S. J.; Brett, M.; Wilson, J.; Millman, K. J.; Mayorov, N.; Nelson, A. R. J.; Jones, E.; Kern, R.; Larson, E.; Carey, C. J.; Polat, İ.; Feng, Y.; Moore, E. W.; VanderPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E. A.; Harris, C. R.; Archibald, A. M.; Ribeiro, A. H.; Pedregosa, F.; van Mulbregt, P.; and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17: 261–272.
- Wang, S.; Teng, Y.; and Perdikaris, P. 2020. Understanding and mitigating gradient pathologies in physics-informed neural networks. arXiv:2001.04536.
- Yuksel, S. E.; Wilson, J. N.; and Gader, P. D. 2012. Twenty Years of Mixture of Experts. *IEEE Transactions on Neural Networks and Learning Systems*, 23(8): 1177–1193.

TECHNICAL APPENDIX

Presented here is supplementary technical information. Please see the original paper for the main presentation of our results. Videos showing the adaptivity and evolution of the POU cells and spline knots during training for Problem 3 and Problem 4 are included as part of the multimedia supplementary appendix.

Training Parameters

Training parameters and hyperparameters for each of the four problems are described below.

Problem 1

We aim to recover via regression the function

$$f(x) = \sin(2\pi x) \quad x \in [0, 1].$$

We employ a dataset of 1000 points, sampled from $[0, 1]$ using a random uniform distribution. We construct a separate validation dataset (also used for plotting) of another 1000 points, independently randomly uniformly sampled from $[0, 1]$.

Random NumPy calls are seeded by using a NumPy random number generator with `seed=1234`. TensorFlow randomness is set via a global random seed of `seed=1234`, which is independent of the NumPy generator and is restarted each time a new network is trained.

Our polynomial-spline networks are constructed with spatial dimension $d = 1$, polynomial degrees $B \in \{0, 1, \dots, 6\}$, and number of splines in the spline layer $N_{\text{splines}} \in \{4, 8, 16, 32, 64\}$. For each value of N_{splines} , we test with the number of POU cells $N_{\text{cells}} = \{1, 2, 4, 8, \dots, N_{\text{splines}}\}$. Our model is constructed with a LSGD layer which uses an L_2 regularizer of 10^{-10} as part of the least-squares solve, to protect against numerical instability in TensorFlow's backwards differentiation of the least squares function call.

During training, we use the Adam optimizer (Kingma and Ba 2014) with a learning rate of 5×10^{-3} , with a batch size of 1000, i.e. performing gradient descent, not stochastic gradient descent. We train for 500 epochs, minimizing mean squared error as our loss function.

Problem 2

We aim to recover via regression the function

$$f(x) = |\sin(3\pi x^2)| + |\cos(5\pi x^2)| \quad x \in [0, 1].$$

Our polynomial-spline networks are constructed with spatial dimension $d = 1$, polynomial degrees $B \in \{0, 1, 2, 3\}$, and number of splines in the spline layer $N_{\text{splines}} \in \{4, 8, 16, 32, 64\}$. For each value of N_{splines} , we test with the number of POU cells $N_{\text{cells}} = \{2, 4, 8, \dots, N_{\text{splines}}\}$. Our model is constructed with a LSGD layer which uses an L_2 regularizer of 10^{-12} as part of the least-squares solve, to protect against numerical instability in TensorFlow's backwards differentiation of the least squares function call.

All other parameter and hyperparameter choices are identical to those listed in Problem 1.

Problem 3

We aim to solve the boundary-value partial differential equation

$$\begin{aligned} -d^2 u &= 2 && \text{on } \Omega = [0, 1] \\ u &= 0 && \text{at } \partial\Omega = \{0, 1\}. \end{aligned} \quad (7)$$

To do so, we minimize the Euler-Lagrange loss

$$L(u) = \int_{\Omega} \frac{1}{2} \|\nabla u\|^2 dx + \beta (u(0)^2 + u(1)^2). \quad (8)$$

Random seeding for TensorFlow initialization is set the same way as in the first two problems. We use the Adam optimizer, enhanced with LSGD, enforced as a layer in the TensorFlow graph, with a L_2 regularizer of 10^{-8} . The penalty parameter β in the Euler-Lagrange loss is set to $\beta = 1000$.

For $B = 0$, we use a polynomial-spline network with $N_{\text{splines}} \in \{4, 8, \dots, 64\}$ and with fixed $N_{\text{cells}} = 3$. We train for 1000 epochs using the Adam optimizer with an initial learning rate of 0.01, reducing to 0.005 halfway through training. For $B = 1$, we use a polynomial-spline network with $N_{\text{splines}} = 5$ and $N_{\text{cells}} = 3$. we train for only 100 epochs, since we recover the true solution faster as it lies in our solution space, and we only use the reduced learning rate of 0.005.

Problem 4

We aim to solve the problem

$$\begin{aligned} -\Delta u &= 0 && \text{on } \Omega = [-1, 1] \times [0, 1] \\ \partial_n u &= 0 && \text{on } \Gamma_N = [-1, 0] \times \{0\} \\ u &= g(r, \theta) && \text{on } \Gamma_D = \partial\Omega \setminus \Gamma_N. \end{aligned} \quad (9)$$

To do so, we minimize the Euler-Lagrange loss

$$L(u) = \int_{\Omega} \frac{1}{2} \|\nabla u\|^2 dx + \beta \int_{\Gamma} (u - g(r, \theta))^2 ds. \quad (10)$$

Random seeding for TensorFlow initialization is set the same way as in the first two problems. We use the Adam optimizer, enhanced with LSGD, enforced as a layer in the TensorFlow graph, with a L_2 regularizer of 10^{-8} . The penalty parameter β in the Euler-Lagrange loss is set to $\beta = 1000$.

We use a polynomial-spline network with $B = 1$, with $N_{\text{splines}} = 9$ for both the x-axis splines and y-axis splines (recall our spline basis for $d > 1$ is formed via a tensor product of 1D splines), and with fixed $N_{\text{cells}} = 16$. We train for 3000 epochs using the Adam optimizer with an initial learning rate of 0.01, reducing to 0.005 halfway through training. Additionally, we re-initialize the optimizer after every 500 epochs, as there was a noticeable improvement in training when doing so.

We plot using 1600 points randomly uniformly sampled from $[0, 1]^2$. These points are not used during training. Contour plots are generated using the PyPlot package's `tricontourf` function, using default parameters for setting the contours.

Notes on LSGD Implementation

We expedite the training of our models by using least-squares gradient descent (LSGD) methods (Cyr et al. 2020) as part of our optimization strategy. Following the approach from (Cyr et al. 2020), we define the function $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^{N_{\text{cells}} d_P}$ as $\Phi : x \mapsto \left(\sum_{\gamma} w_{\alpha, \gamma} \phi_{\gamma}(x) \right) p_{\beta}(x)$, and rewrite our model as

$$y(x) = c^T \Phi(x)$$

for a vector of coefficients $c \in \mathbb{R}^{N_{\text{cells}} d_P}$. LSGD adds a least-squares solve for c between each gradient step of the first-order optimizer.

At this point, we have two options for how to insert this least-squares solve into our training routines. Following (Cyr et al. 2020), we can incorporate this least-squares solve as a TensorFlow Callback (e.g. via NumPy calls to solve the linear system) as a set of operations independent of updating the TensorFlow graph. Alternatively, we can embed the least-squares solve directly into the TensorFlow graph as a TensorFlow Layer, thereby directly enforcing that all outputs of the network lie on the manifold of best-fit solutions regarding the coefficients of the outermost layer.

Note that there are practical differences in using the LSGD callback implementation vs. adding the least-squares step as a layer to the TensorFlow graph. Using the callback implementation of LSGD enables accuracy up to machine precision, but in practice training takes more iterations until convergence, since our iterates after each full training step no longer live on the manifold of least-squares solutions with respect to the basis constructed by the inner layers. In contrast, the layer implementation directly enforces that the output of the network lies on this manifold, and as a result training generally converges faster in practice, but the accuracy of the computation is limited by the stability of the backwards-differentiability of the least-squares solver in e.g. TensorFlow, which is substantially less accurate than machine precision. In our experiments, we could only achieve $O(10^{-8})$ accuracy with the LSGD layer, as compared to $O(10^{-20})$ or better with the callback. To demonstrate, we repeat the hp -convergence test for Problem 1; the results of using the TensorFlow Callback vs. TensorFlow Layer implementations are seen in Figure 8.

For code implementations of both the TensorFlow Layer and TensorFlow Callback implementations, please see the supplementary code appendix.

Notes on Spline Layer Implementation

To build our polynomial-spline network, we build B1-spline basis functions for our domain $\Omega \subset \mathbb{R}^d$ as a tensor product of B1-spline basis functions along each dimension. For each dimension, we construct a B1-spline layer, whose knots are parameterized to accommodate TensorFlow backwards differentiation during training.

We outline our implementation of 1-dimensional B1-spline basis functions on the interval $[0, 1]$. These basis functions are fully described by their knots $\{0 = t_0, t_1, \dots, t_{N-1}, t_N = 1\}$. These knots must remain ordered, else the basis functions will no longer form a POU, nor will they necessarily remain a basis.

In what follows, let $\sigma : \mathbb{R}^N \rightarrow [0, 1]^N$ be the softmax function. We parameterize a B1-spline layer in TensorFlow with a vector $\mu \in \mathbb{R}^N$, where for $i = 1, \dots, N$, the values $\sigma(\mu)_i$ define the interval between knots t_{i-1} and t_i . Additionally, define the matrix $A \in \mathbb{R}^{N \times n}$ as the lower-triangular matrix of all 1's, that is,

$$A_{ij} = 1 \quad \text{if } i \geq j.$$

From the vector μ and the matrix A we can then recover the vector of knots via the expression

$$t_0 = 0; \quad t_{1:N} = A\sigma(\mu).$$

This method for expressing the knots has several advantages. First, we ensure that our knots remain ordered: by definition,

$$\begin{aligned} t_{i+1} &= (A\sigma(\mu))_{i+1} \\ &= \sum_{j=1}^{i+1} \sigma(\mu)_j \\ &= \sum_{j=1}^i \sigma(\mu)_j + \sigma(\mu)_{i+1} \\ &= t_i + \sigma(\mu)_{i+1} \\ &> t_i, \end{aligned}$$

where the last inequality is strict due to the softmax function always being positive. Second, by the same reasoning, the interval widths always remain positive as well, ensuring that our B1-spline basis functions never truly collapse to zero. As a result, we can always define quadrature points within the support of this hat function, ensuring the stability of our integration routines. Third, we can enforce convex combinations so that our knots span our domain, without needing to enforce inequality or even equality constraints, performing unconstrained optimization during training instead of constrained optimization to enforce that our splines remain valid, even as the spline knots evolve during training.

To see how the splines evolve while the knots remain valid, see the multimedia supplemental submissions. In `slit_domain_resampled.gif`, the solution to Problem 4 is plotted after each epoch (resampled for one frame every 10 epochs), and along each axis the corresponding splines that form the B1-spline tensor product basis are plotted as well. As training continues, we see the splines along the x -axis slowly being pulled towards the singularity at the point $(0.5, 0)$, which is where the slit domain ends (after our remapping for Ω). Similarly, in `poisson1d_training_resampled.gif`, the solution to Problem 3 is displayed at each epoch; looking at the evolution of the intervals in the derivative plots, we see the mesh first adapting to resolve the boundary conditions, pushing the intervals towards the endpoints $x = 0$ and $x = 1$, and then later during training pulling intervals back towards the center, becoming nearly uniformly spaced by the end of training.

For code that implements this layer, please see the supplementary code appendix.

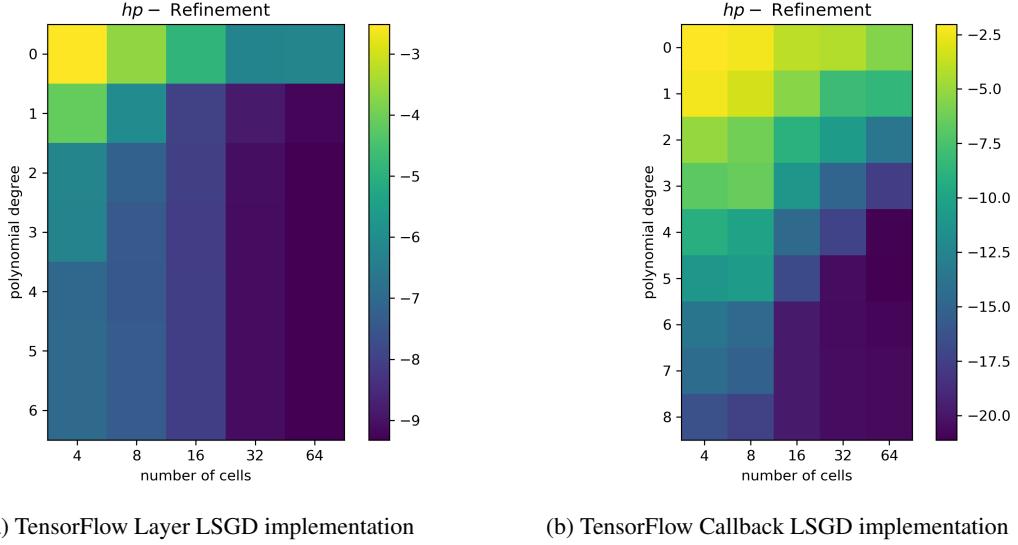


Figure 8: Comparison of $hp-$ convergence results using TensorFlow Callback vs. TensorFlow Layer implementations of LSGD.

Supplementary Results: Problem 3, $B \geq 1$

We repeat our experiments from before using $B = 1$, since in the case $B \geq 1$, the true solution to Problem 3 lies in our solution space. Training details are described above.

After training for only 100 epochs, we recover an L_2 error of 0.009, which is better than the results obtained by training our $B = 0$ models for significantly more epochs. Results are shown in Figure 9. In the error plot on the right, we see that there is error is dominated by enforcing the boundary conditions, since these are enforced by penalty instead of variationally (as a Finite Element method would do). Even though we are not using mean squared error as our loss, we recover an MSE of 2.7252×10^{-6} , which is close to our observed limitations from the stability of the automatic differentiation of the Cholesky decomposition in the linear solve as part of the LSGD implementation.

FEM Comparisons

Our finite element comparisons for Problem 4 are implemented via FEniCS. Our mesh generation for the non-adaptive problems, and the starting mesh for the adaptive problem, are uniform triangular meshes, created via FEniCS’s function `UniformSquareMesh`. We employ P1 Lagrange finite elements as our approximation space upon the constructed meshes. Our errors are computed by overintegrating with a quadrature 3 degrees higher than necessary.

For the non-adaptive comparisons, we use the default values for the linear solver using the `solve` function. For the adaptive problem, we use a mesh energy function $J = \int_{\Omega} u^2 dx$ to indicate where refinement is necessary. Adaptivity is then computed using the built-in `AdaptiveLinearVariationalSolver` function, using default parameters except for the linear dual variational solver for computing the error control function, which is set

to use the conjugate gradient method. We perform adaptive mesh refinement until we reach a solver tolerance level of 0.002, which is the closest step to where the problem size on the refined mesh is similar to that of the linear solve in the polynomial-spline network.

We plot the finite element comparisons on a uniform triangular mesh of size 100×100 , again with the mesh generated by the FEniCS function `UniformSquareMesh`.

Code for solving these variational problems via finite elements is included in the supplementary material.

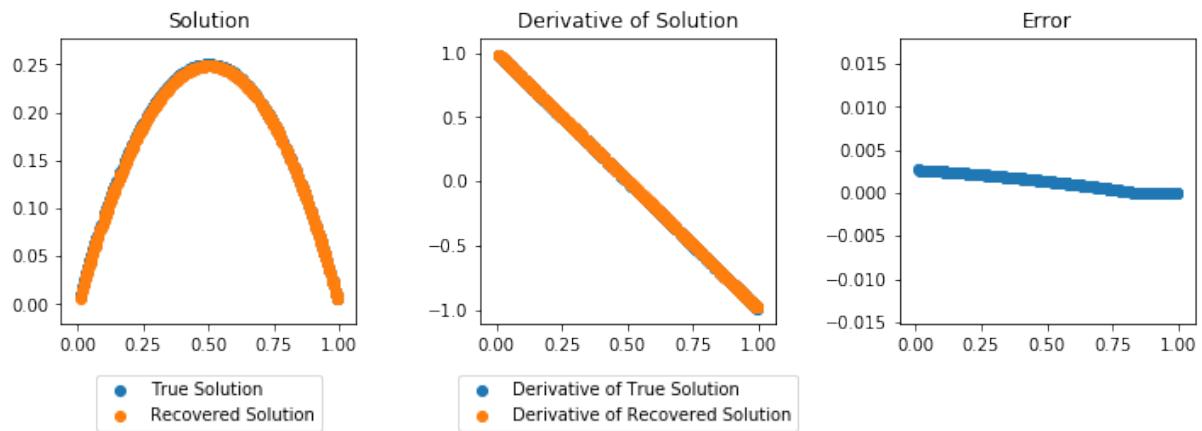


Figure 9: Results for $B = 1$ for Problem 3, which should recover our solution exactly. We see that some error remains, due to enforcing the boundary conditions via penalty and due to the stability of the automatic differentiation of the linear solve in LSGD.