

Free University of Berlin

Department of Mathematics and Computer Science Institute of
Mathematics

master's thesis

Machine learning assisted optimization of
stabilization parameters for
convection-diffusion-reaction equations

submitted by: Vincent Schormann
matriculation number: 5097352
supervised by: Prof. Dr. Volker John

Contents

1	Introduction	2
1.1	The SUPG method	3
1.2	SUPG parameter choice as an optimization problem	5
1.2.1	A model problem	6
1.2.2	Choice of an appropriate loss function	8
2	Formulation as a deep learning problem	12
2.1	The forward pass	12
2.2	The backward pass	13
2.3	Parameter update	14
2.3.1	Mini batch gradient descent	14
2.3.2	Adaptive learning rate	14
2.4	Interfacing Pytorch and FEniCSx: a self-supervised approach	15
2.5	A supervised learning formulation of the problem	17
3	Deep learning architectures	19
3.1	The multilayer perceptron	19
3.1.1	Fully connected layers	19
3.1.2	MLPs for predicting SUPG parameters	20
3.2	Convolutional neural networks	21
3.2.1	Convolution kernels as filters	21
3.3	Graph convolutional neural networks	23
4	Numerical experiments	25
4.1	SPDE 1	27
4.2	SPDE 2	35
4.3	SPDE 3	36
4.4	SPDE 4	44
4.5	SPDE 5	45
4.6	SPDE 6	47
4.7	SPDE 7	51
4.8	Discussion and outlook	53
5	Revisiting the supervised approach	55
5.1	Issues encountered in the previous approaches	55
5.1.1	Definition of an appropriate loss functional	55
5.1.2	Scope of the model	56
5.1.3	Model size and architecture	58
5.1.4	Optimization loop	59
5.2	Results of the revised models	59
6	Appendix	62

1 Introduction

Approximating solutions to singularly perturbed convection-diffusion-reaction equations with standard methods, such as the Galerkin method, can lead to unphysical behavior, including spurious oscillations. Stabilized finite element methods, such as the SUPG method, aim to mitigate these effects by incorporating weighted-residual terms into their bilinear and linear forms. It is known which weightings are admissible to guarantee a unique solution to the approximation problem, and heuristic values are used in applications; however, in general, an optimal choice of weights in two or more dimensions is not known. Choosing SUPG parameters that are too small does not resolve the oscillations, and picking them too large can lead to solution smearing.

One approach is to model the choice of parameters as an optimization problem, where the search space is a subspace of the admissible parameters and a suitable objective function is chosen to identify the optimal solution as the one that minimizes this function. In [16], John, Knobloch, and Savescu present an optimization of SUPG parameters for different objective functions using the BFGS algorithm. Yadav and Ganesan modified this approach in [33] and [31], employing neural networks. Later, they presented a model based on a convolution neural network in [32] using FEniCS [1, 21] and Pytorch [22] that can handle more complex meshes. A recently submitted paper [23] by Petr Knobloch and Manoj Prakash presents a more refined approach that yields the best prediction results so far.

The models presented to date use local data to predict the SUPG parameter for a given cell. Although Yadav and Ganesan present their ConvStab network [32] as a convolutional neural network, it is unclear whether the model incorporates data from adjacent cells. From the context, it can be deduced that it doesn't, and that the ConvStab network is a local multilayer perceptron, same as the SPDE-Net [33] and SPDE-Net II [31]. This thesis proposes a machine learning model based on the graph attention network (GATv2) architecture [6] that incorporates data from adjacent cells to predict local SUPG parameters. It will show that this semi-local approach can yield improved results for heterogeneous datasets.

The description of the foundations of deep learning will mostly follow the book [5] by Chris and Hugh Bishop. The finite element computations are performed using FEniCSx: Solvers are implemented with DOLFINx [3], elements using Basix [25, 24], and variational forms, including symbolic computations on them, use UFL [2]. The neural networks are implemented in PyTorch [22] and PyTorch Geometric [10]. The code is hosted in a public GitHub repository at https://github.com/vschormann/MA_ML_assisted_SUPG.

The goal is to find a suitable approximation of the scalar convection-diffusion-reaction equation. Let $\Omega \subset \mathbb{R}^d$, $d \in \{1, 2, 3\}$, be a bounded polyhedral domain with Lipschitz boundary, $\varepsilon > 0$, $\mathbf{b} \in W^{1,\infty}(\Omega)^d$, $c \in L^\infty(\Omega)$ and $f \in L^2(\Omega)$. The boundary of Ω is called $\Gamma = \overline{\Gamma^D} \cup \Gamma^N$, where Γ^N and Γ^D are relatively open and disjoint, and $\text{meas}(\Gamma^D) > 0$. With $u_D \in H^{1/2}(\Gamma^D)$ and $g \in L^2(\Gamma^N)$, a scalar convection-diffusion-reaction equation is given by

$$-\varepsilon \Delta u + \mathbf{b} \cdot \nabla u + cu = f, \quad (1a)$$

$$u|_{\Gamma^D} = u_D, \quad (1b)$$

$$\varepsilon \frac{\partial u}{\partial n} |_{\Gamma^N} = g. \quad (1c)$$

It is assumed that $\exists c_0 \in \mathbb{R}$ so that

$$0 \leq c_0 \leq c - \frac{1}{2} \operatorname{div} \mathbf{b}, \quad (2)$$

and that the inflow boundary lies completely in $\overline{\Gamma^D}$

$$\{\mathbf{x} \in \Gamma : (\mathbf{b} \cdot \mathbf{n})(x) < 0\} \subseteq \overline{\Gamma^D}.$$

Equation (1) is convection-dominated if $\varepsilon \ll \|\mathbf{b}\|_{L^\infty(\Omega)}$.

In order to understand the term singularly perturbed, one can consider ε as a perturbation term. If the limit of the ε -dependent solution of (1)

$$\lim_{\varepsilon \rightarrow 0} u_\varepsilon$$

or one of its derivatives tends to a discontinuous limit, large gradients form in thin regions called layers and lead to numerically ill-conditioned discrete problems when standard methods are applied. The SUPG method is more robust but requires selecting appropriate parameters to achieve satisfactory results.

1.1 The SUPG method

The streamline upwind Petrov-Galerkin method, introduced by Brooks and Hughes in [14], is also known as the streamline diffusion finite element method (SDFEM). The monograph [26] by Hans G. Roos, Martin Stynes, and Lutz Tobiska provides a comprehensive review of its theory. The aspects that are relevant to this thesis are repeated here without proof. Let V_h and W_h be subspaces of a Lagrange finite element space P with evaluation points $\mathbf{DOF} = x_1, \dots, x_N$. The solution space W_h contains functions that approximate the Dirichlet boundary conditions

$$W_h = \left\{ w \in P : w(x_j) = u_D(x_j), \forall x_j \in \mathbf{DOF} \cap \overline{\Gamma^D} \right\},$$

while the test space V_h contains functions, whose values on $\overline{\Gamma^D}$ approximate homogeneous boundary conditions

$$V_h = \left\{ v \in P : v(x_j) = 0, \forall x_j \in \mathbf{DOF} \cap \overline{\Gamma^D} \right\}.$$

The residual terms of the SUPG method are defined per cell and multiplied by a scalar weighting term. This weighting is a finite element function $y_h \in Y_h$ of order 0, defined on the same mesh as W_h and V_h . Let $(\cdot, \cdot)$ be the L^2 -scalar product, Δ_h the cell-wise defined Laplace operator, and

$$\begin{aligned} a(u, v) &= \varepsilon(\nabla w \cdot \nabla v) + (\mathbf{b} \cdot \nabla w, v) + (cw, v), \\ s_h(y_h; w, v) &= (-\varepsilon \Delta_h w + \mathbf{b} \cdot \nabla w + cw, y_h \mathbf{b} \cdot \nabla v), \\ r_h(y_h; v) &= (f, y_h \mathbf{b} \cdot \nabla v), \\ R(y_h; w, v) &= a(w, v) + s_h(y_h, w, v) - (f, v) - (g, v)_{\Gamma^N} - r_h(y_h, v). \end{aligned} \quad (3)$$

If there exists a unique solution $u_h = u_h(y_h) \in V_h$ for $y_h \in Y_h$, such that

$$R(y_h; u_h, v_h) = 0 \quad \forall y_h \in Y_h, \quad (4)$$

we call u_h the solution of the SUPG approximation or just a solution if the context is unambiguous.

Let K denote a cell in the mesh of the decomposition of Ω ; then a sufficient condition for the existence of a unique solution of the SUPG approximation is

$$0 \leq y_h(\mathbf{x}) \leq \frac{1}{2} \min \left\{ \frac{h_K^2}{\varepsilon c_{inv}}, \frac{c_0}{\|c\|_{L^\infty(K)}^2} \right\}, \quad (5)$$

if $c|_K \not\equiv 0$. The quantities h_K and c_{inv} are chosen so that the inverse inequality

$$\|\Delta w\|_{0,K}^2 \leq \frac{c_{inv}}{h_K} \|\nabla w\|_{0,K}^2 \quad \forall w \in W_h \quad (6)$$

is satisfied, c_{inv} depends only on W_h . The term h_K is a characteristic element dimension for K that only depends on the shape of K . In [13], Harari and Hughes give explicit representations and values for h_K and c_{inv} for triangular and rectangular elements of orders 1 to 3. In the case of constant convection $\mathbf{b} \in \mathbb{R}^d$ and without reaction term $c|_K \equiv 0$, the weaker condition

$$0 \leq y_h(\mathbf{x}) \leq \frac{h_K^2}{\varepsilon c_{inv}^2}. \quad (7)$$

is a sufficient condition for the existence of a unique solution.

A standard heuristic for choosing SUPG parameters is:

$$\tau_K = \frac{h_K}{2\|\mathbf{b}\|_K} \xi(Pe_K), \quad Pe_K = \frac{\|\mathbf{b}\|_K h_K}{2\varepsilon}, \quad (8)$$

where ξ is called the upwind function, Pe_K is the local Péclet number, $\|\mathbf{b}\|_K$ is a suitable norm for $\mathbf{b}$, such as $\|\mathbf{b}\|_{L^\infty(K)}$. The function ξ is often chosen as

$$\xi_0(\alpha) = \coth(\alpha) - \frac{1}{\alpha}$$

or an approximation of ξ_0 , namely

$$\xi_1(\alpha) = \max \left\{ 0, 1 - \frac{1}{\alpha} \right\}, \quad \text{or} \quad \xi_2(x) = \min \left\{ 1, \frac{\alpha}{3} \right\}.$$

The definition of h_K in [8] need not coincide with that for [5]. A standard and easy-to-implement choice is the diameter of K . In [15], it was shown that the diameter in the streamline direction $diam(K, \mathbf{b})$ can yield improved results over this choice.

For simplicity, the constraint

$$0 \leq y_h|_K \leq 100\tau_K \quad (9)$$

can be used instead of [5] and we will call a solution with $y_h|_K = \tau_K$ a solution obtained with standard parameters or u_h^{std} . A solution of the Galerkin approximation will be called $u_h^{Galerkin}$.

To assess the quality of an approximation, analytical solutions are available for only a few examples. In most cases, we have to rely on a posteriori error estimates as described in [28]. Those estimates also provide a starting point for defining the loss function in [16].

In this thesis, only linear triangular and bilinear rectangular Lagrange elements, i.e., $\mathcal{P}_1$ or $\mathcal{Q}_1$ will be considered. In those spaces, [6] is always satisfied as the Laplacian vanishes in each cell. For problems without a reaction term, the only constraint is that the parameters are positive.

1.2 SUPG parameter choice as an optimization problem

Let $I : W_h \rightarrow \mathbb{R}$ be a continuous functional whose global minima correspond to optimal solutions, and let $u_h(y_h)$ be a solution of the SUPG approximation for a given problem. We aim to solve the following optimization problem:

$$\text{minimize}_{y_h} \Phi(y_h) = I(u_h(y_h)), \quad (10a)$$

$$\text{subject to } R(y_h; u_h, v) = 0 \quad \forall v \in V_h. \quad (10b)$$

The constraint for this problem is a weak form of a partial differential equation and can thus be classified as an optimal control problem. Under suitable conditions, it can be solved using gradient descent or quasi-Newton methods. The gradient can be efficiently computed using the adjoint method as demonstrated in [16]:

$$(\partial_w R)'(y_h; u_h, v)[\psi] = DI(u_h) \quad \forall v \in V_h, \quad (11)$$

$$\Phi(y_h) = -(\partial_{y_h} R)'(y_h; u_h, \psi). \quad (12)$$

Here, the adjoint operator for a real bilinear form $A(u, v)$ with test function v and trial function u , changes the order of input: $A'(u, v) = A(v, u)$, and a Fréchet derivative with respect to a coefficient is evaluated in the argument written in square brackets

$$\begin{aligned} \partial_v A(u, v) : V_h &\rightarrow \mathbb{R}, \\ v_0 &\mapsto \partial_v A(u, v)[v_0]. \end{aligned}$$

Consequently, two linear problems have to be solved to compute the gradient:

1. the SUPG approximation for u_h ,
2. the adjoint problem to compute ψ .

UFL can symbolically compute the Fréchet derivatives of R and I and thus can implement these computations independently of the chosen loss function I .

Several loss functions are discussed in [16]. If an analytical solution u is available, $I^{L^2}(u_h) = \|u_h - u\|_{L^2(\Omega)}^2$ can be used. Residual-based functionals are also considered:

$$\begin{aligned} I^{\text{cross}}(u_h) = \sum_{K \in \mathcal{T}_h, \bar{K} \cap \Gamma^D = \emptyset} & (\| -\varepsilon \Delta u_h + \mathbf{b} \cdot \nabla u_h + cu_h - f \|_{0,K}^2 \\ & + \|\phi(\mathbf{b}^\perp \nabla u_h)\|_{L^1(K)}), \quad (13) \end{aligned}$$

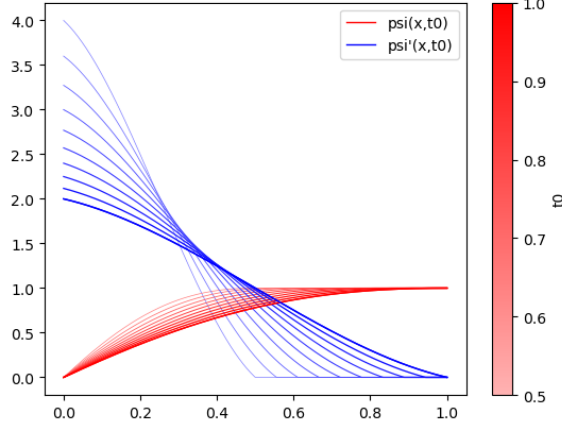


Figure 1: $\Psi(x, t_0)$ and $\Psi'(x, t_0)$ for $0 \leq x \leq 1$ and $1/2 \leq t_0 \leq 1$.

where

$$\mathbf{b}^\perp = \begin{cases} \frac{(b_2(\mathbf{x}), -b_1(\mathbf{x}))^T}{|\mathbf{b}(\mathbf{x})|} & \mathbf{b}(\mathbf{x}) \neq 0, \\ 0 & \mathbf{b}(\mathbf{x}) = 0, \end{cases}$$

and

$$\phi(x) = \begin{cases} \sqrt{x} & x \geq 1, \\ 0.5(5x^2 - 3x^3) & x < 1. \end{cases}$$

In [20], those results are modified by concatenating the cellwise loss terms of the residual function with a function Ψ that satisfies $\Psi(0) = 0$ and $\Psi(t) = 1$ for a chosen parameter $0 < t_0 \leq t$ in order to address more general problems:

$$I^{lim}(u_h) = \sum_{K \in \mathcal{T}_h, \overline{K} \cap \Gamma^D = \emptyset} \Psi(\| -\varepsilon \Delta u_h + \mathbf{b} \cdot \nabla u_h + cu_h - f \|_{0,K}^2), \quad (14)$$

$$\Psi(t) = \xi\left(\frac{t}{t_0}\right),$$

$$\xi(x) = \begin{cases} \frac{1}{2}x^4 - x^3 - \frac{1}{2}x^2 + 2x & x \leq 1, \\ 1 & x > 1. \end{cases}$$

The term I^{cross} dampens large values in the residual and defines a threshold where every cell in which the residual exceeds t_0 is excluded in the computation of the gradient. This is due to the fact that the derivative of Ψ vanishes for any argument larger than t_0 as shown in Figure 1.

1.2.1 A model problem

Consider the following convection diffusion equation on the unit square $\Omega = (0, 1)^2$:

$$-10^{-8} \Delta u + \partial_x u = 1 \quad \text{in } \Omega, \quad (15a)$$

$$u|_{\partial\Omega} \equiv 0. \quad (15b)$$

A physical solution to this problem is close to $u = x$ in most of Ω and exhibits sharp layers at the outflow boundary $x = 1$ and at the lateral boundaries $y \in \{0, 1\}$

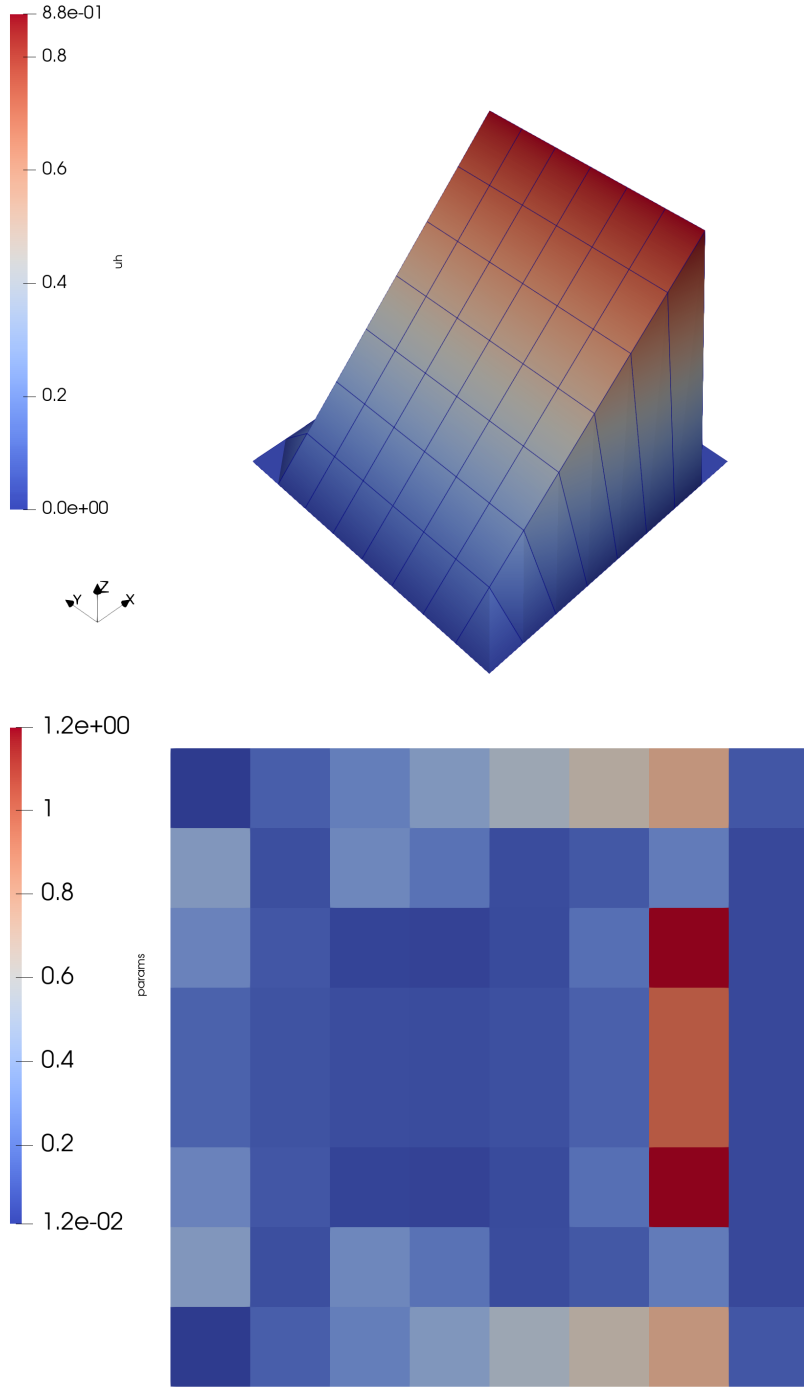


Figure 2: Problem (15) on a uniform, quadrilateral 8 by 8 mesh with optimized SUPG Parameters obtained from a minimization with the L-BFGS-B algorithm with box constraints and I^{cross} as an objective function.

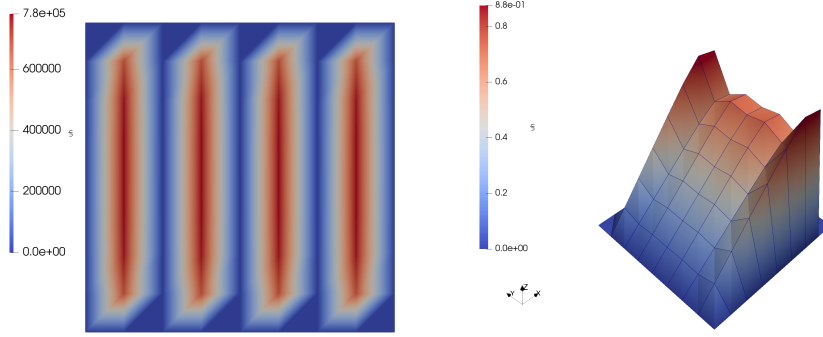


Figure 3: Solutions of the Galerkin approximation $u^{Galerkin}$ (left) and an SUPG u^{std} approximation with standard parameters.

away from the inflow boundary $x = 0$ due to the homogeneous boundary conditions. More specifically, since $\Delta u = (\partial_x u - 1)/\varepsilon$, a good approximation is expected to have large Δu_h wherever $\partial_x u_h \not\approx 1$. This effect is most pronounced close to the outflow boundary due to the sign change of $\partial_x u_h$ and subsequent exponential decay of u_h in streamline direction. This layer is called exponential boundary layer, and the layers at the lateral boundaries are called parabolic boundary layers.

Note that for $\mathcal{P}_1$ and $\mathcal{Q}_1$, $\Delta u_h|_K$ vanishes. A large Δu should therefore result in large jumps of ∇u_h across facets.



Figure 2 shows an approximate solution of (15) where the SUPG parameters were optimized using the L-BFGS-B algorithm, which is a quasi-Newton method approximating the BFGS algorithm. This algorithm can handle constraints, such as (4), as demonstrated in 7. For $m \in \mathbb{N}$, it stores the parameters and gradients of the last m iterations and uses them to compute an approximation of $H_f^{-1} \nabla f$. As an objective function, the sum of the residual and crosswind term I^{cross} was used.

Approximations on the same mesh obtained using the standard Galerkin method exhibit large spurious oscillations, with function values exceeding 10^5 as shown in Figure 3. If standard SUPG parameters are chosen, the oscillations in streamline direction vanish. In the crosswind direction, however, the solution still exhibits oscillatory behavior and a disparity between values in cells near the lateral boundaries and in interior cells. Additionally, the sign change of ∇u_h in the exponential boundary layer causes only a small jump of ∇u_h across the boundaries, and thus the solution is smeared in the exponential boundary layer.

1.2.2 Choice of an appropriate loss function

Consider the cellwise residual of a diffusion reaction equation. Since the Laplacian vanishes for first-order finite elements, the term is equal to

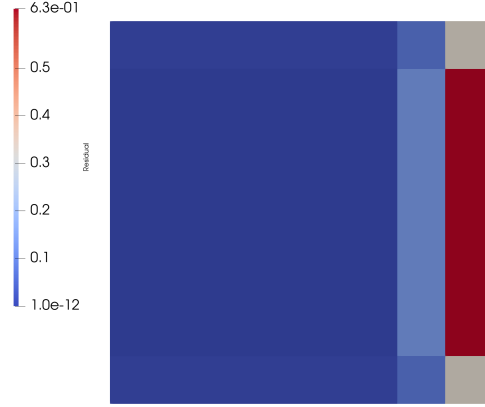


Figure 4: Residual of a solution of (15) with optimal parameters.

$$\sum_{K \in \mathcal{T}_h} \|\mathbf{b} \cdot \nabla u_h - f\|_{0,K}^2.$$

An optimal solution of (15) with respect to this objective function will show an error in the exponential boundary layer, regardless of how fine the mesh or accurate the solution. As the solution needs to tend to zero near the boundary, it must deviate from $u(x, y) \approx x$, which is penalized by the residual. In [28], Verfürth modified this estimator by incorporating the jumps of ∇u_h across the edges, which approximates Δu_h , but an optimal solution is prevented from producing jumps of magnitude $(\nabla u_h \cdot \mathbf{b} - f)/\varepsilon$ when $h_K \gg \varepsilon$ unless spurious oscillations are introduced. Consequently, the optimal solution deviates from $u_h \approx x$ only on the cells at the boundary, where a linear function connects the exposed edges to $u_D \equiv 0$. This produces a jump of approximately $1/h_K$ in the exponential boundary layer near $x = 1$.

In [16], the jump terms were dropped, as their impact on the error is marginal. Additionally, the cells at the Dirichlet boundary are omitted, since errors there cannot be resolved by an optimal solution on a reasonably fine mesh, and the solution in those cells is completely prescribed by the boundary condition and the point values of neighboring cells.

Unphysical behavior in crosswind direction, as shown in Figures 3, doesn't lead to large errors in the residual, as it only considers the streamline direction and thus I^{cross} was introduced, which is a suitable error indicator for (15). However, this error indicator is not ideal for all problems.

Consider a convection diffusion reaction with $\varepsilon = 10^{-3}$, $\mathbf{b} = (2, 3)^T$ and f, u_D chosen such that

$$u = 16x(1-x)y(1-y) \left(\frac{1}{2} + \frac{\arctan[2\varepsilon^{-1/2}(0.25^2 - (x-0.5)^2 - (y-0.5)^2)]}{\pi} \right)$$

solves (1). The analytical solution u is a cylinder surrounded by an interior layer. The source term f prescribes a steep increase in u_h at the layer's influx and a corresponding decrease at its outflux. The L_2 -projection of u in Q_1 on an 8×8 quadrilateral mesh, together with the source term and the convection field, can be

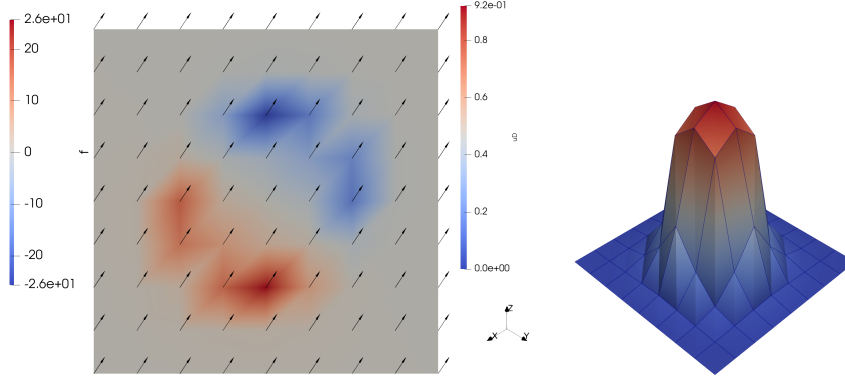


Figure 5: Source and convection term $f, \mathbf{b}$ (left) and Q_1 -interpolation of analytical solution u (right) of an example with interior layers on a quadrilateral 8×8 mesh.

seen in Figure 5.

In order to force ∇u_h to have the desired magnitude, the numerical solution takes on negative values. This is exacerbated by ignoring boundary cells, particularly in meshes with few cells. Figure 6 shows an approximate solution with SUPG parameters obtained from an optimization with respect to I^{cross} on the same mesh used earlier and a solution with parameters optimized with respect to I^{L_2} . The approximate solution on the left has a loss value $I^{cross}(u_h) \approx 8.4$ and the exact solution $I^{cross}(u) \approx 11.1$, even though the behavior of u_h is clearly unphysical. This demonstrates that I^{cross} is not a suitable loss functional for this problem. Better results can be obtained by including boundary cells, using I^{lim} , or using I^{L_2} as the loss function.

To date, no universal loss function is known that yields optimal results across all problems. Consequently, one must be careful to select an appropriate loss function for each problem when designing a neural network.

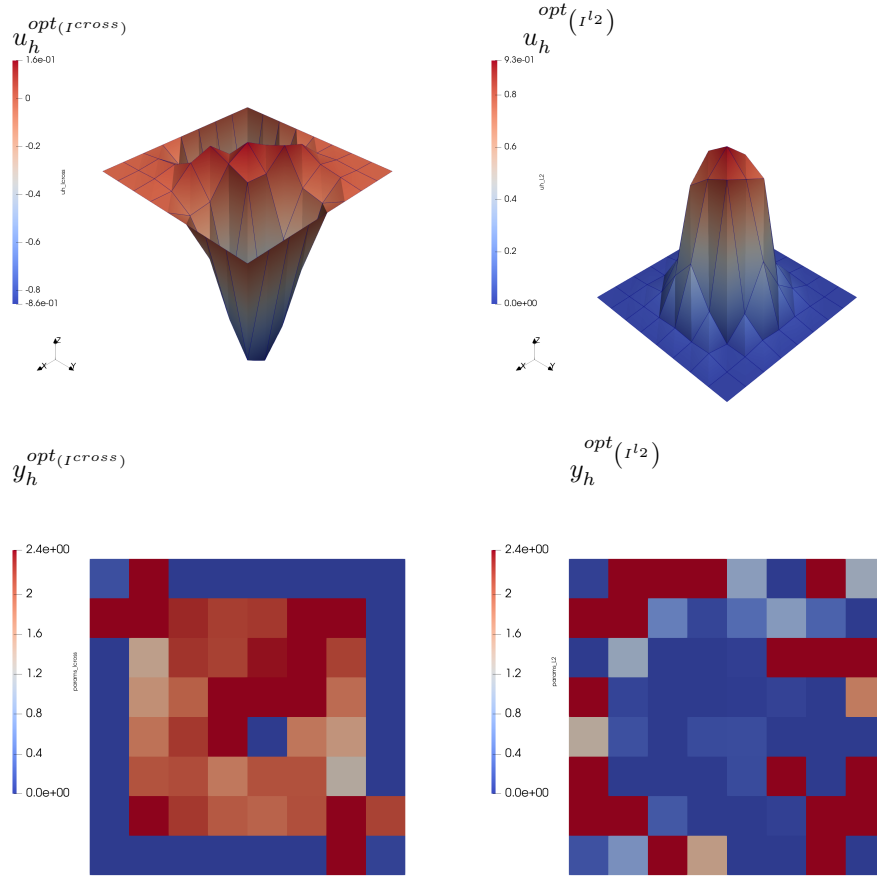


Figure 6: , and approximate solution u_h optimized with respect to I^{cross} . Loss values: $I^{cross}(u_h) \approx 8.4$ and the exact solution $I^{ross}(u) \approx 11.1$.

2 Formulation as a deep learning problem

The design process for a deep learning problem, given an optimization problem, is not straightforward. The underlying optimal control problem described in Section 1.2 can be interpreted as a simple model that encodes the problem data and the triangulation in the input format and the loss function, respectively. It thus lacks the flexibility to handle any problem other than the one it is trained on. The main functionalities of a model are prediction and learning. The implementation of the former is called a forward pass, whereas learning is made possible through the implementation of a backward pass.

In PyTorch, a tensor is a multidimensional array of real-valued elements, such as 32-bit floating-point numbers. It is stored and processed on a device, such as a CPU or GPU, and can be configured to hold additional data, such as gradient information. A tensor that stores gradient information is a set of learnable parameters (weights). Let w be a one-dimensional tensor of weights and let x be an object that we will call input. Training a model aims to find optimal values for w , so w must be fixed in size. As shown in Section 3, certain architectures support inputs of varying size and structure.

2.1 The forward pass

A trained neural network nn has fixed weights and provides an output $nn(x, w)$ given an input x . This mapping is called the forward pass. The details of its implementation make up the network architecture, which will be discussed in the next section. For now, we will only assume that there exist functions $f^0, \dots, f^{L-1}$ and some integers $1 = i_0 \leq i_1 \leq \dots \leq i_{L-1} = \dim(w)$, $w^k := (w_{i_{k-1}}, w_{i_{k-1}+1}, \dots, w_{i_k})$ so that the forward pass can be expressed as:

$$nn(w, x) = [f^L(w^L) \circ f^{L-1}(w^{L-1}) \circ \dots \circ f^1(w^1)](x). \quad (16)$$

We say that the network has depth L and that

$$z^k := [f^k(w^k) \circ \dots \circ f^1(w^1)](x)$$

is the output of the k -th layer. We define the input x as z^0 and rewrite the last equation as $z^k = f^k(w^k, z^{k-1})$. The final output predicts the values of y_h . More precisely, if $v_1, \dots, v_{\dim(Y_h)}$ is a global basis of Y_h , $nn(w, x) = z^L$ are the coordinates of y_h :

$$y_h = \sum_{i=1}^{\dim(Y_h)} z_i^L v_i. \quad (17)$$

Since Y_h is a real vector space, the model output is continuous, and thus the prediction task can be classified as a regression problem.

Defining the input x as the output of a trivial zeroth layer hints that its structure depends on the network's architecture. In many deep learning models, x will be a tensor. A three-dimensional tensor of size $(3, W, H)$, for example, can represent a colored image with W horizontal times H vertical pixels, where each pixel has a scalar value for its red, green, and blue channels, respectively. A more general data structure is a simple graph with one-dimensional tensors of size C_{in} associated with each node. We will see that this definition can be interpreted as a generalization of the previous example.

It is possible to pass non-tensor arguments to a network. However, this has serious drawbacks: PyTorch is designed to learn tensor parameters and thus relies on tensor operations for training.

2.2 The backward pass

During training, gradient descent will be used to adjust the weights w . Let X be the set of admissible inputs for the model, W the space of possible weights, and $E : X \times W \rightarrow \mathbb{R}_+$ be a loss function. An approach to measuring the quality of the model for a certain $w \in W$ would be to compute

$$\int_X E(nn(x, w)) dx.$$

For infinite sets, it is not possible to evaluate all possible inputs, and additionally, we cannot guarantee that X is even measurable. To avoid this issue, we will sample from X and evaluate finite subsets of X .

Let X_0 be a finite subset of X called the training set, then

$$\sum_{x \in X_0} \frac{E(nn(x, w))}{|X_0|} \quad (18)$$

is called the training error. Minimizing the training error is called fitting the training data, and the model's ability to fit a wide array of training data is called its representational capability.

To quantify the model's quality, a second data set, X_1 , called a test set, can be defined. The test error can be computed analogously to (18). The model's ability to predict correct results for data not contained in the training set is called generalization.

In Section 1.2, we already saw how to compute the gradient for an optimization problem constrained by (4). However, optimization of neural networks involves additional computations called backpropagation.

This depiction of the backpropagation algorithm is taken from [11] and assumes only that the functions $\{f^k\}_{k=1}^L$ are differentiable. Calculating the derivatives with respect to the $\{z^k\}_{k=1}^L$ can be performed using the chain rule and (16). Let $J_{f^k}(z^k)$ denote the Jacobian matrix of the vector valued function $f^k(w^k, \cdot)$ at z^{k-1} and $J_{f^k(z^{k-1})}(w_k)$ the Jacobian of $f^k(\cdot, z^{k-1})$ at w^k , then

$$D_{z^k} E(nn(w, x)) = \left(\prod_{i=0}^{L-k} J_{f^{L-i}(w^{L-i})}(z^{L-i-1}) \right) \nabla E(z^L)$$

and

$$D_{w^k} E(nn(w, x)) = (J_{f^k(z^{k-1})}(w_k)) D_{z^k} E(nn(w, x)).$$

Modern deep learning libraries such as PyTorch do not require users to implement the backward pass manually, provided the forward pass uses tensor operations.

2.3 Parameter update

With the derivatives $D_w E(w, x)$ for $x \in X$ from the backward pass, the derivative with respect to the training X_0 can be computed:

$$D_w E(w) := \frac{1}{|X_0|} \sum_{x \in X_0} D_w E(nn(x, w)).$$

This can be used to implement gradient descent $w_i := w_{i-1} - lr D_w E(w)$ for a fixed learning rate $lr > 0$. In practice, there are modifications to the training loop that have proven effective for deep neural networks. The most important modifications are mini batch gradient and adjusting the learning rate during training.

2.3.1 Mini batch gradient descent

In most cases, it is not feasible to directly compute the derivative for the whole data set. Consider, for example, the popular dataset Fashion-MNIST [30], which contains 60.000 28×28 grayscale images, making a single update computation costly. A practical alternative is to sample batches X_b of size N from X_0 and update the weights according to

$$w \leftarrow w - \frac{lr}{N} \sum_{x \in X_b} D_w E(nn(x, w)). \quad (19)$$

This method is called mini-batch gradient descent. For $N = 1$ this is equivalent to a stochastic gradient descent and for $N = |X_0|$ it is equivalent to standard gradient descent. For this reason, standard gradient descent is also referred to as full batch gradient descent. An iteration in which the entire dataset passes through the network, and the weights are updated using (19) for each batch, is called an epoch.

2.3.2 Adaptive learning rate

Consider a model $nn(w, x) = nn(x) = x$, $E(x) = 10.000x^2$, $x = 1$ with a learning rate of 10^{-4} , then $E'(x) = 20.000$ and after an update according to (19), we get $x \leftarrow 1 - \frac{20.000}{10^4} = -1$. After another update, $x \leftarrow 1$. This training loop will not converge unless the learning rate is reduced. The learning rate can be reduced automatically using a step scheduler, such as StepLR, which multiplies the initial learning rate by a factor $0 < \gamma < 1$ after a fixed number of epochs. Additionally, optimization methods that adapt the learning rate depending on previously computed data are available.

In 2011, Duchi, Hazan, and Singer introduced the AdaGrad algorithm [9], which reduces the learning rate for each parameter w_i individually by using the accumulated sum of squares of all the derivatives for that parameter:

$$\begin{aligned} r_i^0 &= 0, \\ r_i^\tau &= r_i^{\tau-1} + \left(\frac{\partial E(z^L)}{\partial w_i} \right)^2, \\ w_i^\tau &= w_i^{\tau-1} - \frac{lr}{\sqrt{r_i^\tau} + \delta} \left(\frac{\partial E(z^L)}{\partial w_i} \right), \end{aligned}$$

where δ is a small positive parameter preventing division by 0. The RMSProp multiplies the terms in the second row with weights $0 < \beta < 1$ and $(1 - \beta)$.

The most commonly used algorithm to schedule the learning rate is the Adam algorithm, which was introduced by Kingma and Ba in 2014 [17]:

$$\begin{aligned} s_i^0 &= 0, \\ r_i^0 &= 0, \\ s_i^\tau &= \beta_1 s_i^{\tau-1} + (1 - \beta_1) \left(\frac{\partial E(z^L)}{\partial w_i} \right), \\ r_i^\tau &= \beta_2 r_i^{\tau-1} + (1 - \beta_2) \left(\frac{\partial E(z^L)}{\partial w_i} \right)^2, \\ \hat{s}_i^\tau &= \frac{s_i^\tau}{1 - (\beta_1)^\tau}, \\ \hat{r}_i^\tau &= \frac{r_i^\tau}{1 - (\beta_2)^\tau}, \\ w_i^\tau &= w_i^{\tau-1} - lr \frac{\hat{s}_i^\tau}{\sqrt{\hat{r}_i^\tau} + \delta}, \end{aligned}$$

with $0 < \beta_i < 1$ for $i = 1, 2$.

As with gradient computation, PyTorch can automatically adjust the learning rate with torch.optim provided that w is a tensor.

2.4 Interfacing Pytorch and FEniCSx: a self-supervised approach

If the forward pass uses a different programming library, the backward pass cannot be computed automatically and must be implemented manually. For the optimal control problem (10) with an objective function $\Phi : Y_h \rightarrow \mathbb{R}_+$, the computation of the gradient requires the assembly of sparse matrices based on finite element methods, which is usually not a task deep learning libraries are designed to perform. Finite element libraries such as FEniCSx can handle these computations more efficiently.

In the case where $\Phi = E$ and $\{f^i\}_{i=1}^L$ are implemented as differentiable functions in PyTorch, it suffices to implement a function that takes z^L as input and returns $\Phi \left(\sum_{i=1}^{\dim(Y_h)} z_i^L v_i \right) = E(z^L)$ in a forward pass and $D\Phi \left(\sum_{i=1}^{\dim(Y_h)} z_i^L v_i \right) = \nabla E(z^L)$ in a backward pass.

In Pytorch, this can be done by subclassing torch.autograd.Function and overwriting the functions forward and backward.

This method requires a tensor input z^L , which is the final output of the model, and a solver object. The solver stores instances of the three linear problems described in Section 1.2 and can update them for a given SUPG parameter choice. It is implemented in FEniCSx and can solve for u_h , $I(u_h) = \Phi(y_h)$ and $D\Phi(y_h)$. Note that the method set_weights() updates all three linear solver objects at once.

The function to_vector in step 1 creates a copy of a torch.tensor in a vector data structure that can be handled by FEniCSx, and the function to_tensor is the inverse operation of that function. Both data structures are mathematically equivalent, as

Algorithm 1 forward

Input: z^L , $\text{FEM}_{\text{solver}}$
1: $z^{\text{vec}} \leftarrow \text{to_vector}(z^L)$
2: $\text{FEM}_{\text{solver}}.\text{set_weights}(z^{\text{vec}})$
3: $u_h \leftarrow \text{FEM}_{\text{solver}}.\text{solve}()$
4: $\mathcal{I}(u_h) \leftarrow \text{FEM}_{\text{solver}}.\text{loss}()$
5: $E(z^L) \leftarrow \text{to_tensor}(\Phi(y_h))$
6: **return** $E(z^L)$

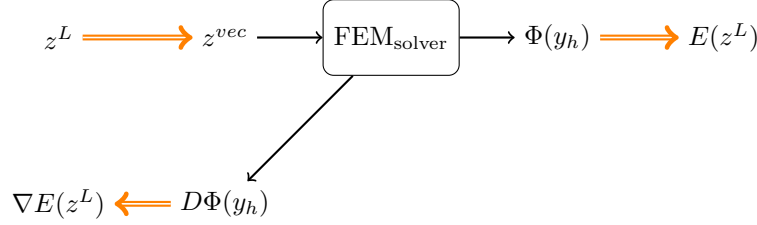


Figure 7: FEM-solver object as described in Algorithms 1 and 2.

they represent the same underlying array. However, the tensor structure is optimized for deep learning tasks such as automatic differentiation, and the vector data structure is optimized for linear algebra tasks needed to solve finite element problems.

In practice, these functions do not immediately output the other format; instead, they leverage the fact that both libraries provide a NumPy interface and can either convert their array structures into an ndarray or share memory with an ndarray. Moreover, many deep learning routines benefit from GPU acceleration, whereas FEniCSx calculations typically run on a CPU. In that case, the `to_vector` function needs to copy objects between different devices.

Algorithm 2 backward

Input: $\text{FEM}_{\text{solver}}$, $J = I_{\dim(z^L)}$
1: $D\Phi(y_h) \leftarrow \text{FEM}_{\text{solver}}.\text{compute_derivative}()$
2: $\nabla z^L \leftarrow \text{to_tensor}(D\Phi(y_h))$
3: **return** $J\nabla z^L$

The backward pass can recover the solver object from a context object stored in the `torch.autograd.Function` instance. Since the linear problem solvers were updated all at once in step 2 of the forward pass, the $\text{FEM}_{\text{solver}}$ object can immediately solve for $D\Phi(y_h)$ and return the result after another transition from a FEniCSx vector to a `torch.Tensor`. Figure 7 shows the data flow described above.

Note that, compared with the approach in [32], which passes the data four times between PyTorch and FEniCS, the algorithms described above transfer the data only three times between PyTorch and FEniCSx. Figure 8 shows the data flow described by Yadav and Ganesan. This approach does not require the FEM solver to perform symbolic computations, since PyTorch’s autograd handles them. Algorithms 1 and 2 save one transformation and make use of the fact that UFL can represent the derivative $D\mathcal{I}(u_h)$ as a symbolic expression with argument u_h , which can be stored in the $\text{FEM}_{\text{solver}}$ -object.

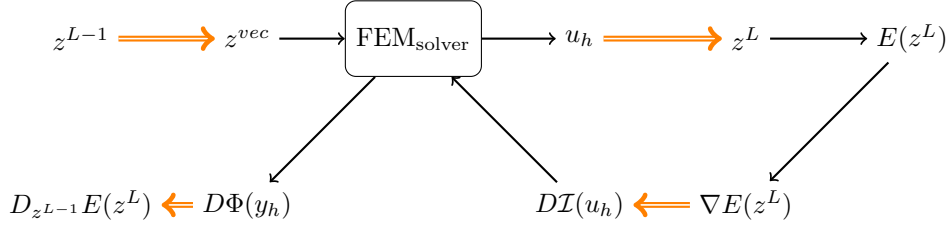


Figure 8: FEM-solver as described in [32].

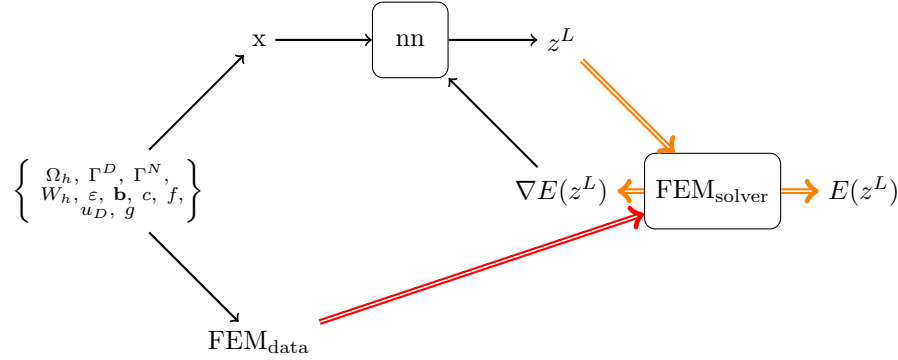


Figure 9: Forward and backward passes given a mesh and a convection-diffusion-reaction equation.

Moving data can create a performance bottleneck, especially when moving between CPU and GPU, as discussed in [8], so reducing the number of data transforms is an important performance consideration. Neither approach relies on validation data provided separately, such as previously computed optimal output values. Therefore, it is a form of self-supervised learning.

2.5 A supervised learning formulation of the problem

With the approach described above, the problem data (1) and the mesh still need to be provided separately to the FEM solver object. Figure 9 shows the transformations necessary to run a forward pass and a backward pass given a convection-diffusion-reaction equation. The intermediate steps in the tensor-to-vector data transformations are collapsed, but the orange color is retained to indicate their presence.

In the first step, the problem data must be encoded as input to a neural network and simultaneously as objects interpretable by FEniCSx. The latter can involve symbolic expressions of functions for which a tensor encoding is not straightforward. In the next step, the input is passed to the neural network, and the finite-element data objects are used to initialize the FEM solver. The solver relies on highly problem-specific data structures, such as sparsity patterns for linear-system matrices or mappings that associate mesh entities with Dirichlet or Neumann boundary conditions. To ensure efficient computation, most Python objects are wrappers around immutable, compiled C++ objects; thus, iterating over many different problems requires repeatedly initializing new solvers. This creates a bottleneck, as highlighted by the red arrow.

Some measures can be taken to alleviate this bottleneck.

Caching: Initializing all solver objects at once prevents multiple solver initializations. This is very effective for small datasets that can be completely loaded into memory. However, it is important to keep in mind that memory is a valuable resource in deep learning, and limiting it can restrict a neural network’s size and capacity.

Optimizing the finite-element code: Recompile only the objects necessary for each iteration over the dataset. Multiple problems defined on the same mesh share the same sparsity pattern in the matrices of their linear systems, allowing us to update data more efficiently than with varying triangulations and domains. It is thus possible to update the SUPG parameters in an already initialized solver at considerably lower cost than initializing a new one.

In general, we can compute many more gradients in less time by minimizing the number of solver initializations. This can be done by computing optimal parameters for each data point individually by solving (10). The model is then trained using the loss function:

$$\|nn(x, w) - t\|_2^2, \quad (20)$$

where $t \in Y_h$ is the optimal SUPG parameter choice for the problem associated with x .

Since the optimal values for the training set are known in advance, this method is a form of supervised learning. A supervised model can be trained in a fraction of the time required for a self-supervised model. On the other hand, a single parameter adjustment in a neural network affects its output across all inputs. Provided that the gradient with respect to a mini-batch contains information that can be generalized to a significant portion of admissible inputs, fewer adjustments might be needed to achieve satisfactory results with a self-supervised approach.

3 Deep learning architectures

Several well-known building blocks in neural network architectures can enhance the model's representational capacity and generalization. All of them impose varying restrictions on the structure of the input x .

3.1 The multilayer perceptron

Let $d_i, d_o \in \mathbb{N}$, $\{\varphi_j\}_{j=1}^{d_i}$ be basis functions that map from the input space to a so-called embedding space, and let ϕ be a nonlinear function mapping from the embedding space to the solution space. An abstract formulation of a perceptron is given by a weighted sum of the basis functions, with possibly a bias term added:

$$y_i = \phi \left(b_i + \sum_{j=1}^{d_i} w_{i,j} \varphi_j(x) \right), \quad i = 1, \dots, d_o.$$

The function y_i is called a neuron. When considering $(\varphi_1(x), \dots, \varphi_{d_i}(x))^T$ as input, this transformation is called a fully connected layer.

3.1.1 Fully connected layers

This layer applies an affine linear transformation to the input, producing a weighted sum

$$z = x^T A + b, \quad (21)$$

and applies a nonlinear activation function ϕ to z . The weights are a matrix $A \in \mathbb{R}^{d_i \times d_o}$ and a vector $b \in \mathbb{R}^{d_o}$ where d_i is the number of in features and d_o is the number of out features. Typical examples for non-linear activation functions are the rectified linear unit (ReLU):

$$\phi(x) = \begin{cases} 0 & x < 0, \\ x & x \geq 0, \end{cases}$$

the logistic sigmoid function:

$$\phi(x) = \frac{1}{1 + e^{-x}},$$

or the Leaky ReLU function:

$$\phi(x) = \begin{cases} 0.01x & x < 0, \\ x & x \geq 0. \end{cases}$$

In the case where $d_o > 1$, the functions are to be understood elementwise.

In the case of a single-layer perceptron, $\varphi_j(x) = e_j \cdot x$. To increase the representational capability of this model, the paradigm of learning basis functions rather than selecting them, when possible, has proven useful in deep learning. A straightforward way to approximate φ_j is to use another fully connected layer:

$$\varphi_i(x) = \phi \left(\tilde{b}_j + \sum_{k=1}^{\tilde{d}_i} \tilde{w}_{j,k} x_k \right).$$

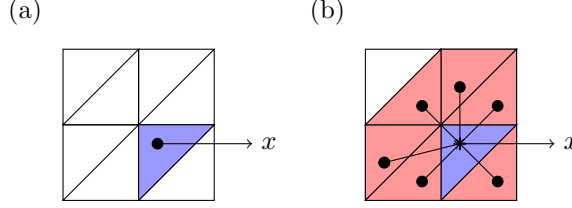


Figure 10: Input data for a neural network. The local approach (a) considers only data from a single cell, while (b) incorporates data from adjacent cells.

This model is called a two-layer perceptron. Applying the same argument repeatedly leads to the multilayer perceptron (MLP). This mechanism of adding depth to a network to increase its representational capacity justifies the term ‘deep learning’.

Many MLPs consist only of concatenated fully connected layers. However, adding depth does not necessarily increase accuracy, as greater degrees of freedom make the model more prone to overfitting. Additionally, due to the chain rule, the gradient of many subsequent layers is a product of multiple factors, which often leads to gradients vanishing or exploding, viz., tending to either zero or ∞ .

3.1.2 MLPs for predicting SUPG parameters

The SPDE-NetII from [31] implements an MLP that assumes constant convection and takes four input arguments ε , b_1 , b_2 , h , where h denotes the area of the cell called mesh size. In the numerical experiments, only uniform triangular meshes are considered, with h constant across all cells. Local phenomena in the data are attended to by dividing the output by the cellwise norm of $\nabla u_h^{Galerkin}$. For $\mathcal{P}_1$, this is just the Euclidean norm, as it is constant, and $\|\nabla u_G\|_K = \|\nabla u_G(c)\|_K$ where c is the centroid of the cell, can be used for non-linear elements.

The SPDE-NetII(local) from the same paper and the ConvStabNet from [32] allow variable inputs for each cell and incorporate $\|\nabla u_h^{Galerkin}\|_K$ as a fifth input parameter. These models can handle variable coefficients and aim to learn a transformation depending on $\|\nabla u_h^{Galerkin}\|$ instead of prescribing $y(x, w)/\|\nabla u_h^{Galerkin}\|$. This approach yields better results than the standard solution and SPDE-NetII across several error metrics for the examples discussed in the paper. This demonstrates the potential to predict SUPG parameters from suboptimal solutions using local data alone. It is noteworthy, however, that the provided visualizations still exhibit layer smearing for most examples.

All models discussed so far are local in the sense that the prediction for $y_h|_K$ depends on a feature vector that is extracted from cellwise data only, as shown in Figure 10 (a). This makes them highly flexible across different problems and mesh structures, since the feature vector can be easily extracted and the output is invariant with respect to mesh geometry, cell ordering, and other global data. Unphysical behavior, such as spurious oscillations and smearing, is, however, often defined by how the solution changes between cells. It is therefore worth exploring how information from adjacent cells can be incorporated, as illustrated in Figure 10 (b).

3.2 Convolutional neural networks

In a quadrilateral mesh, the cells are arranged in a grid structure. For this special case, convolutional neural networks (CNNs) are a well-known architecture that can attend to adjacent data points and are flexible regarding the number and size of cells.

Let x be a 2-dimensional array of shape (H_x, W_x) and let $H_k < H_x$, $W_k < W_x$, then there are $(H_x + 1 - H_k) \cdot (W_x + 1 - W_k)$ overlapping, 2-dimensional slices

$$x[i : i + H_k, j : j + W_k]$$

of shape (H_k, W_k) . A simple form of a 2-dimensional convolution operation applies the same fully connected layer to all of these patches. We say that the layer shares the weights over all slices. This can be implemented by defining a matrix in which each row is a flattened slice, and the flattened kernel is a left-multiplying vector. Note that, since the slices overlap, some data must be copied, and the resulting matrix is larger than the original input vector. This method is called `im2col` or `unfolding`. When both arguments are transposed and their order in the matrix multiplication is reversed, this is called a `transpose convolution`.

Conventionally, an additional batch-channel dimension is assumed. The term `channel` originates from image data formats, in which an RGB pixel has three channels: red, green, and blue. This corresponds to the cell-wise features described in the previous section. Consequently, a kernel for a 2-dimensional convolution has 3 dimensions. 2D convolutional layers in PyTorch also accept 4D inputs when `mini-batching` is used.

Convolution with a kernel of dimensions larger than 1 produces an output smaller than the input, whereas `transpose convolution` produces an output larger than the input. This is called `down-` and `up-sampling`. `Down-sampling` can be prevented by padding the original input vector with rows and columns of zeros at the exposed entries. Assuming that the number of channels stays the same, all kernel dimensions are equal and odd, and padding is chosen to be $(k_{dim} - 1)/2$, then the output will have the same dimensions as the input. Note that a kernel with all dimensions equal to 1 corresponds to the MLP architecture discussed earlier, which is used by `ConvStabNet`.

Excluding slices by skipping a specified number $s \in \mathbb{N}$ of indices each time the kernel moves one position ahead is called a `stride` of $s + 1$. Similarly, when, instead of consecutive indices, only every $\mathbb{N} \ni d$ -th index is considered in the definition of the slices, it is called a `dilation` of size $d + 1$.

3.2.1 Convolution kernels as filters

Let $u : \Omega \rightarrow \mathbb{R}$ be the solution for a convection-diffusion-reaction equation and let $I, J \subset \mathbb{N}$ be index sets such that $(i, j) \in \Omega$ and $i < i+1$, $j < j+1$. For $u_{i,j} := u(i, j)$, the five-point stencil is an approximation for $\Delta u(i, j)$.

$$(\Delta_h u)_{i,j} = \frac{1}{h^2} (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}).$$

If u possesses a sharp edge at (i, j) , the Laplacian is expected to have a high value, and thus, this difference scheme could be used to detect the aforementioned edges, which are known to cause problems in the numerical computation of the solution.

Now let U be a 2-dimensional tensor with elements $u_{i,j}$, then the edges could be detected using a convolution operation with the $(3, 3)$ kernel:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

In image recognition, this technique is called a Laplace filter.

Since the solution is unavailable and the layer's position is not the only factor in predicting SUPG parameters, this filter is unsuitable in this case. The CNN aims to learn a set of filters that fit the provided data and yield positive results.

A trained $(C, 3, 3)$ -shaped kernel can be viewed as a filter that computes a scalar value for the central cell based on its own features and those of the adjacent cells. A single CNN layer requires only a little more data than a local method. With increased depth, however, the information from more distant cells is implicitly included. To see this, consider a sequence of convolutional layers with kernel size $(3, 3)$, using padding of 1 in each layer to preserve the height and width of the original input tensor, while the number of channels may vary. Now let $z_{i,j}^m$ be a point in the output of the m -th layer with $m > 2$, then $z_{i,j}^m$ depends on a slice of z^{m-1} with the same shape as the kernel. Each element of that slice depends on a slice of similar shape in z^{m-2} . The union of those slices has shape $(5, 5)$ and by induction, $z_{i,j}^m$ depends on a slice of shape $(2m+1, 2m+1)$ in z^0 provided $i-m$ and $j-m$ are larger or equal zero and $i+m < H$ and $j+m < W$. This slice in z^0 is called the receptive field of $z_{i,j}^m$.

The receptive field can be further increased by using pooling layers, which partition the input into non-overlapping slices of a specified kernel shape and aggregate the values of each slice. Aggregation is typically a maximum or average function; the maximum function selects the input with the most information to propagate, whereas the average function combines information. This operation loses information and performs downsampling, which is accompanied by an upsampling operation: unpooling. Max unpooling recreates previously down-sampled slice, recovers the element with the maximum value and sets the other values to 0 and average unpooling sets all values to the average.

Figure 11 shows the receptive field of a three-layered CNN with constant kernel sizes of $(3, 3)$ and a stride = dilation = 1 and a max pooling operation with kernel size $(2, 2)$ followed by a max unpooling operation.

Let $C, W, H \in \mathbb{N}$, x be a tensor with shape (C, H, W) , y a CNN with $L-1$ layers and f^L a convolutional or pooling layer with kernel size (H_L, W_L) and let $y(x)_{i,j}$ have a perceptive field of shape (h_{L-1}, w_{L-1}) , then the size of the perceptive field of $f^L(y(x))_{i,j}$ is at least $(h_{L-1}+1, w_{L-1}+1)$ and most $(h_{L-1}^p+2H_L, w_{L-1}^p+2W_L)$. Consequently, for any H, W , one can construct a CNN with enough depth so that each point in the output has x as its perceptive field. Conversely, for every CNN there exists $W, H \in \mathbb{N}$ so that the perceptive field is an arbitrarily small fraction of the input. Moreover, if for any two points $\mathbf{x}_1, \mathbf{x}_2$ in a rectangular domain Ω where x represents a feature map for the cells of a quadrilateral mesh and $y(x)$ predicts the SUPG parameters, a mesh can be chosen fine enough to exclude $\mathbf{x}_1$ from the perceptive field of the cell including $\mathbf{x}_2$.

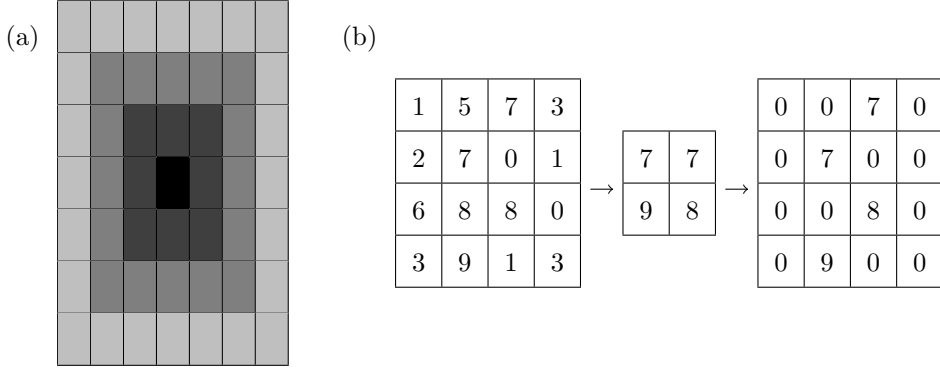


Figure 11: (a) Receptive field of a 3 layer convolutional network with (3,3)-shaped kernels in each layer. (b) Max pooling Operation with kernel size (2,2) followed by max unpooling.

The approach of spatial convolution thus depends on the network’s ability to predict SUPG parameters from cell features and their adjacent cells, where adjacency need not imply geometric proximity.

3.3 Graph convolutional neural networks

The convolutional architectures discussed so far assume that data is organized in a grid structure, which is not always the case. Simple graphs are more flexible and can handle triangular meshes and more complex geometries.

Let G be a simple graph with the adjacency matrix A , where each node corresponds to a cell and an associated feature vector x_K . For $K \in \mathcal{T}_h$ let $\mathcal{N}(K) \subset \mathcal{T}_h$ be the adjacent cells. More precisely, whenever $K, \tilde{K}$ share at least one node, then $A_{K,\tilde{K}} = A_{\tilde{K},K} = 1$. In most cases, self-loops are also inserted, so that each element is adjacent to itself. A graph convolution is an operation that employs a shared weight structure across all cells and is invariant with respect to the number of neighbors $|\mathcal{N}(K)|$ and their ordering. This operation is also called message parsing and has the following general form:

$$x'_K = \gamma_\Theta \left(x_K, \bigoplus_{\tilde{K} \in \mathcal{N}(K)} \phi_\Theta(x_K, x_{\tilde{K}}) \right), \quad (22)$$

where $\gamma_\Theta, \phi_\Theta$ are differentiable and $\bigoplus$ is differentiable and permutation invariant.

The first graph convolutional network (GCN) was introduced by Kipf and Welling in [18]. The convolution operation applies a linear transformation W to a weighted sum of the x_K and its neighbors:

$$x'_K = W \left(\sum_{\tilde{K} \in \mathcal{N}(K) \cup K} \frac{x_{\tilde{K}}}{\sqrt{(|\mathcal{N}(\tilde{K})| + 1)(|\mathcal{N}(K)| + 1)}} \right). \quad (23)$$

This operation has the same number of parameters as a local MLP model but averages over data from a cell and its neighbors. A classic convolution with the same perception window has $(\mathcal{N}(K) + 1)|x_K| = 9|x_K|$ parameters for quadrilateral meshes. It is clear that classic convolution can not assign a parameter to each

neighbor, since it is required to be invariant with respect to $|\mathcal{N}(K)|$.

The expressivity of this operation can be increased by applying one linear transformation W_1 to x_K , and another one W_2 to the vector aggregated from the neighboring cells. This operation is called GraphSAGE and was presented in [12]

$$x'_K = W_1 x_K + W_2 \left(\sum_{\tilde{K} \in \mathcal{N}(K)} \frac{x_{\tilde{K}}}{|\mathcal{N}(K)|} \right). \quad (24)$$

Both (23) and (24) have fewer learnable parameters than a 2D Convolution with a kernel larger than $(1, 1)$. While fewer parameters may reduce expressivity, the message-parsing architecture is invariant to transformations such as rotation. The attention mechanism can compute individual weights $\alpha_{K, \tilde{K}} \in \mathbb{R}$, called attention coefficients, for each $\tilde{K} \in \mathcal{N}(K)$.

Let $x_K \in \mathbb{R}^D$ for all $K \in \mathcal{T}_h$, $D' \in \mathbb{N}$ and $W_s, W_t \in \mathbb{R}^{D' \times D}$, $a \in \mathbb{R}^{D'}$ be learnable parameters. Popular graph attention operations are the graph attention mechanism, GATConv:

$$\alpha_{K, \tilde{K}} = \frac{\exp(\text{LeakyReLU}(a^T (W_s x_K + W_t x_{\tilde{K}})))}{\sum_{\tilde{K} \in \mathcal{N}(K)} \exp(\text{LeakyReLU}(a^T (W_s x_K + W_t x_{\tilde{K}})))}, \quad (25)$$

and the GATv2Conv:

$$\alpha_{K, \tilde{K}} = \frac{\exp(a^T \text{LeakyReLU}((W_s x_K + W_t x_{\tilde{K}})))}{\sum_{\tilde{K} \in \mathcal{N}(K)} \exp(a^T \text{LeakyReLU}((W_s x_K + W_t x_{\tilde{K}})))}. \quad (26)$$

The vector x'_K is computed as the weighted sum of the images $W_t x_{\tilde{K}}$ for all $\tilde{K} \in \mathcal{N}(K)$:

$$x'_K = \sum_{\tilde{K} \in \mathcal{N}(K)} \alpha_{K, \tilde{K}} W_t x_{\tilde{K}},$$

GATv2Conv only exchanges the order of the nonlinear activation *LeakyReLU* and the linear function a^T . The GAT framework was introduced in [27] by Veličković et al. and later refined by Brody et al. in [6], where it was shown that (26) is strictly more expressive than (25), since the latter computes only static attention. More precisely, the nodes of G can be ordered by a global ranking so that if $\text{rank}(\tilde{K}) < \text{rank}(\hat{K}) \Rightarrow \alpha_{K, \tilde{K}} \leq \alpha_{K, \hat{K}}$ for all $K \in G$. In comparison, GATv2Conv enables dynamic attention, allowing each node to attend to its neighbors individually. If a global ordering applies to a given problem, GATConv may be better suited, as it is less prone to overfitting.

4 Numerical experiments

In order to assess which of the architectures described above is best suited for the optimization problem (10), the following general network structure was implemented for each operation: 1 input layer l^1 with 10 input parameters and 5 output parameters, 8 hidden layers $l^2, \dots, l^9$ with 5 input and 5 output parameters and 1 output layer l^{10} with 5 input and 1 output parameters. As a non-linear activation function ϕ , the ReLU function was applied to the output of layers 1 – 9.

We say nn is a network with 10 input parameters and 1 output parameter, depth 10, and width 5. This structure was identified through iterative experimentation starting with a configuration similar to the one proposed in [32], which had 3 hidden layers with widths 64, 32, and 32. The chosen configuration yielded the best results in this case; we do, however, not claim that it is optimal. Here, it mainly serves to emphasize the differences among the layer architectures.

Applying the ReLU activation to the output of the final layer l^{10} yielded unsatisfactory results because most models produced only positive outputs, with the exception of the self-supervised GATv2-model, which took on mostly negative values. Instead, the absolute value of $l^{10}(z^9)$ was chosen, yielding improved results.

As input, a 10-dimensional vector for each cell was created from the function data. Let $\varepsilon, \mathbf{b}, c, f$ define a convection-diffusion-reaction equation, and for any of those terms let $\cdot|_K$ denote the evaluation of that term at the barycenter of the cell K , then the input parameters for K are defined as:

1. the diffusion coefficient: $\varepsilon|_K$,
2. the convection in x -direction: $b_1|_K$,
3. the convection in y -direction: $b_2|_K$,
4. the reaction term: $c|_K$,
5. the source term: $f|_K$,
6. the cell diameter: h_K ,
7. the value of the solution with standard SUPG parameters at the barycenter of K : $u^{std}|_K$,
8. the value of the partial derivative with respect to x of the standard solution at the barycenter of K : $\partial_x u^{std}|_K$,
9. the value of the partial derivative with respect to y of the standard solution at the barycenter of K : $\partial_y u^{std}|_K$,
10. an indicator whether it is a quadrilateral or triangular mesh:

$$v_{10} = \begin{cases} 3 & K \text{ is a triangle,} \\ 4 & K \text{ is a quadrilateral.} \end{cases}$$

All architectures discussed so far have been implemented. Namely, a local MLP, based on (21) and graph networks using GCN (23), SAGEConv (24), GAT (25), and GATv2 (26) convolution operations as layers. As described in Section 3.3, the input for the graph networks also includes an adjacency matrix that connects cells sharing one or more vertices.

Each network was implemented using the self-supervised approach described in Section 2.4, where the same objective function as in (10) is minimized. For a particular layer architecture, the associated model will be denoted by a superscript, e.g., nn^{MLP} . The method for obtaining the standard solution u^{std} , defined by (8), will be denoted by std , and a direct optimization of (10) by opt . Each model was also trained using the supervised approach described in Section 2.5, where (20) was optimized. For the supervised models, an s is added as a subscript to the model, e.g., nn^{MLP_s} . Similarly, we call the prediction of the parameters y_h^{MLP} and the solution u_h^{MLP} for the MLP-model and analogously for the other models.

The training set contains 7 singularly perturbed convection-diffusion-reaction equations on the unit square $\Omega = (0, 1)^2$. These problems will be referred to as a numbered SPDE and described in the subsections that follow. Each problem was implemented on multiple meshes with varying cell sizes and aspect ratios. Since all training problems are defined on the unit square, the mesh configurations can be described by two integers N_x and N_y . For a quadrilateral mesh, Ω is partitioned into N_x equally sized parts along the x -axis and N_y equally sized parts along the y -axis, so that all cells are rectangular, have equal size, and N_x describes the number of cells in the x direction and N_y the number of cells in the y direction. For triangular meshes, a mesh defined by N_x and N_y consists of the quadrilateral mesh defined by the same parameters, where the rectangles are divided from bottom left to top right.

Mesh configurations were chosen based on the problem and the quality of the solution obtained from direct optimization. Quadrilateral meshes were easier to fit for most problems, so all configurations with $N_x, N_y \in 2^j$ and $j \in \{3, 4, 5, 6\}$ were implemented. Examples in which direct optimization failed to produce satisfactory results were removed manually. As most problems proved harder to fit on triangular meshes, more even aspect ratios were chosen for those implementations. More precisely, N_x and N_y were chosen as randomly generated integers between 30 and 45. The same method was used for problems that required more supervision during the optimization.

It is well known that if one uses mesh cells with higher aspect ratios, where the long axes are perpendicular to the layers, one obtains inaccurate results. Long aspects should always align with the layer's direction.

The final dataset contains 88 data points. It proved difficult to fit because it includes problems that require different objective functions to achieve good results, and has triangular and quadrilateral meshes of a wide variety of aspect ratios. In comparison, the models from [31] and [32] only consider problems on uniform meshes where good results can be achieved using I^{cross} .

The test set contains one implementation of each problem on both a quadrilateral and a triangular mesh, with randomly chosen N_x and N_y values between 30 and 45.

The predictions of the models are compared to the standard solution u^{std} with parameters defined by (8) and an optimized solution u^{opt} obtained from optimizing (10) where the search space is constrained by (9). The optimizations were performed using the L-BFGS-B algorithm in SciPy [29] with $gtol$ and $ftol$ set to 10^{-16} . Initially, a limit of 5,000 iterations was set, but some problems required well over 10,000 iterations to yield satisfactory results. For example, problem 3 on a 64×8 quadrilateral mesh required approximately 12,000 iterations, due to the steep edges in the solution and the high aspect ratio. Some problems on triangular meshes didn't converge after more than a million iterations.

The neural networks were optimized for 20.000 epochs using an initial learning rate of 10^{-3} , the Adam algorithm, and the ReduceLROnPlateau learning rate scheduler that reduces the current learning rate by a factor of 0.8 whenever the change in the loss function remains smaller than 10^{-4} for more than 10 iterations. While 20.000 may seem like a relatively small number to train a network, given that individual optimization took comparably long or even longer to converge, it is noteworthy that an epoch computes the gradient for all problems in the training set. Additionally, each network trains parameters that predict a solution locally, so that a single gradient for a network should be understood cell-wise, possibly including adjacent cells. The number of gradients used for training is thus proportional to the total number of cells summed over all meshes in the training set. Consequently, the training progress usually stagnated before reaching the final epoch.

4.1 SPDE 1

Let $\Gamma^D = \partial\Omega$, $\varepsilon = 10^{-8}$, $u_D = 0$, $\mathbf{b} = (1, 0)^T$, $f = 1$. This is (15). The optimized parameters were obtained by minimizing I^{cross} . In the test set, there are two implementations of this problem. One on a quadrilateral mesh with $N_x (= 32) \times N_y (= 30)$ cells and one on a triangular mesh with $2 \times N_x (= 30) \times N_y (= 32)$ cells. The exact configurations for all implementations in the training and test sets with references to the names in the gallery linked above are shown in Figure 43 in the Appendix.

For the quadrilateral mesh, the direct optimization took 45.788 iterations and 48.196 function evaluations overall.

model	$I^{cross}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	0.70	0.0	1.16
<i>opt</i>	$8.89 \cdot 10^{-10}$	0.0	0.97
nn^{MLP}	0.05	0.0	1.01
nn^{GCN}	0.83	-0.03	0.11
nn^{SAGE}	0.69	-0.02	0.63
nn^{GAT}	2.70	0.0	1.10
nn^{GATv2}	$2.66 \cdot 10^{-4}$	0.0	0.97

While none of the models achieve results comparable to direct optimization, nn^{MLP} and nn^{GATv2} yield improvements over the standard solution. The predicted SUPG-parameters are shown in Figure 12, and a warped visualization is added to show the extreme variation in their structure. Notice that nn^{SAGE} improves the objective function value I^{cross} , but the solution u^{SAGE} takes negative, thus unphysical, values and has a low maximum value, indicating a smeared solution. Figure 13 shows that this is in fact the case.

The prediction of nn^{MLP} exhibits an artifact in the lateral layer at $y = 0$. The top images in Figure 14 show the function values of the optimal solution and the best performing models nn^{MLP} and nn^{GATv2} at the lateral boundaries at $x = 1 - 1/N_x \approx 0.97$, the location of the sign change of the exponential boundary. The left figure clearly shows the prediction artifact for nn^{MLP} . The bottom images

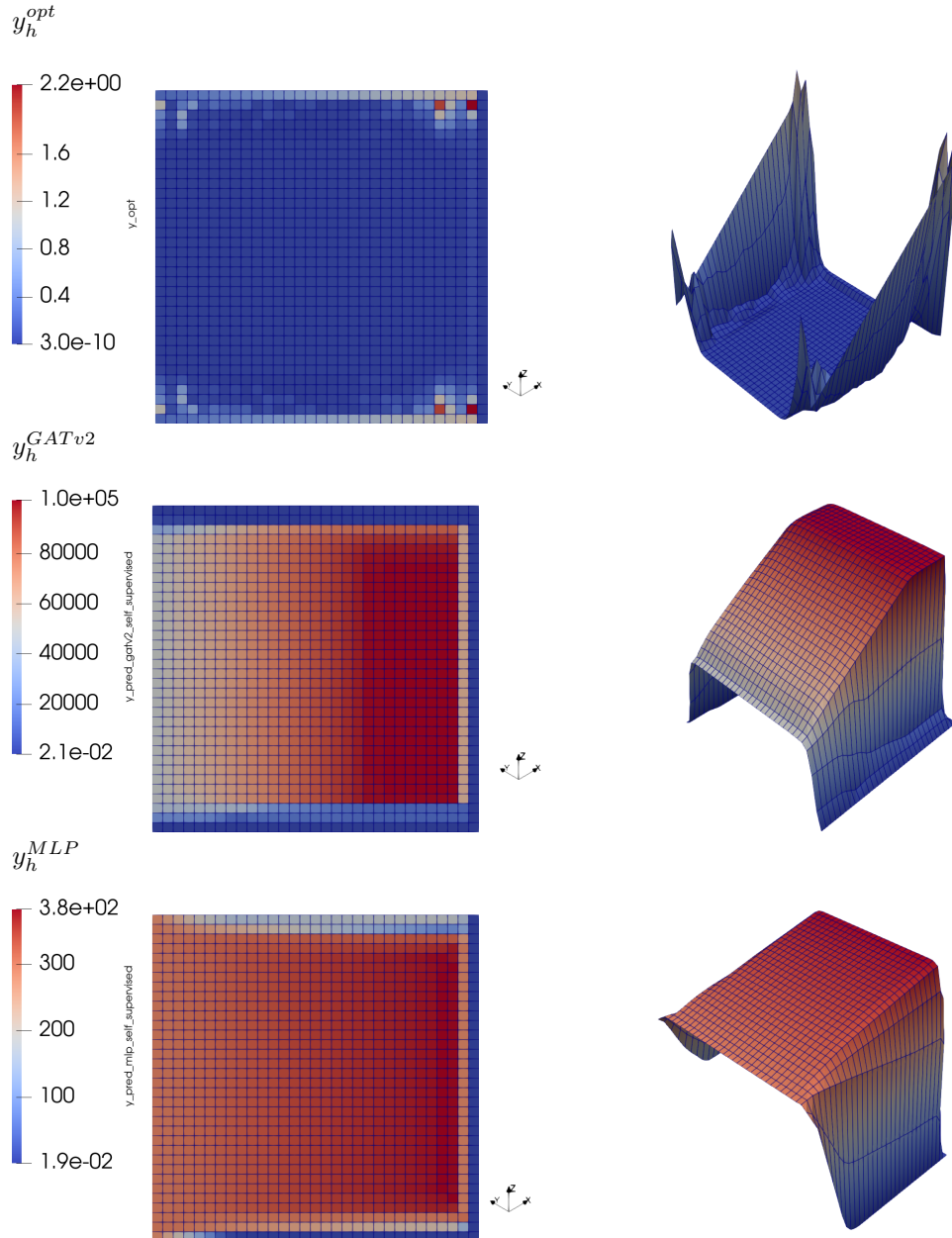


Figure 12: SPDE 1: SUPG-parameters obtained from direct optimization and nn^{MLP} and nn^{GATv2} .

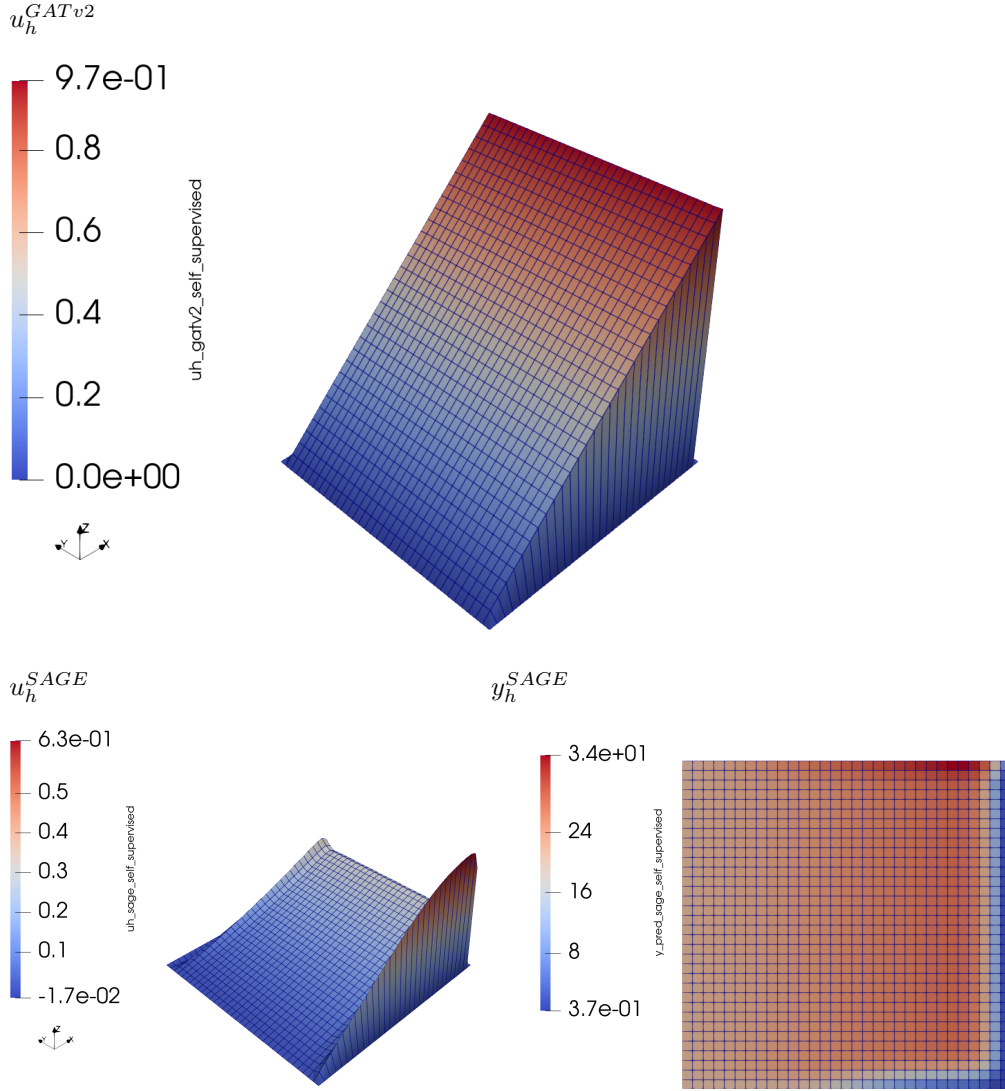


Figure 13: SPDE 1: Solution predicted by nn^{GATv2} , the best performing model for this problem, and solution and parameters of the poor performing nn^{SAGE} .

show the exponential boundary at $y = 0.5$. All images show a plot of $u \equiv x$ for comparison, a function for which the residual and I^{cross} vanish, but that doesn't satisfy the boundary conditions.

It takes considerably longer to find optimal values for this problem on triangular meshes of similar sizes. For a mesh with $2 \times N_x (= 30) \times N_y (= 32)$ cells, the direct optimization was stopped after 2.000.000 iterations.

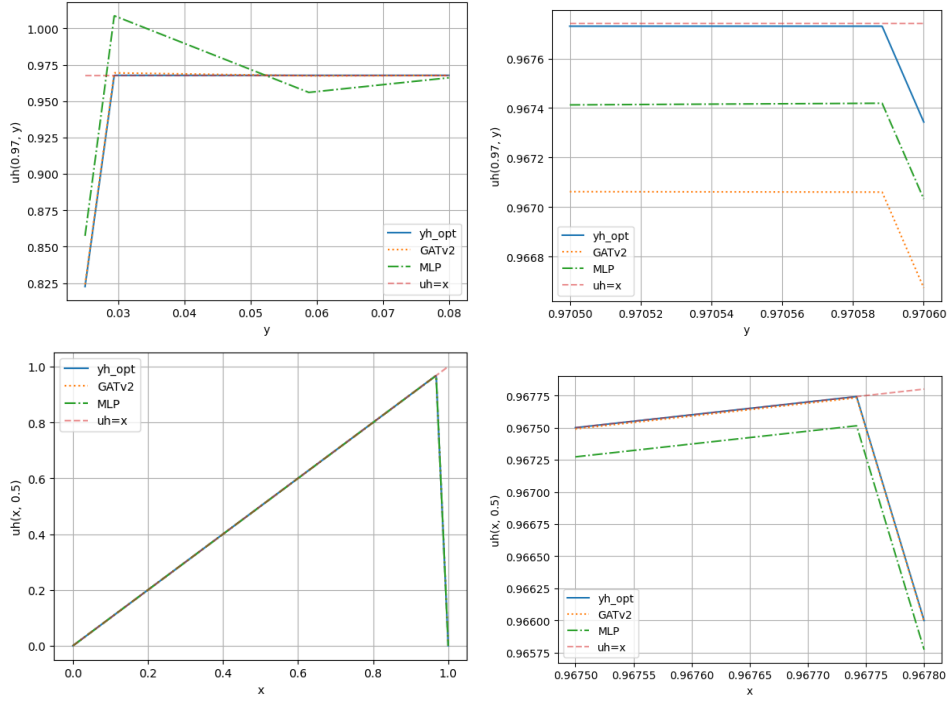


Figure 14: SPDE 1: Function values of the approximations of nn^{MLP} and nn^{GATv2} at the exponential layer (top) $x = 1 - 1/31 \approx 0.97$ and along the streamline (bottom) at $y = 0.5$. The values for $u \equiv x$ are also shown for comparison.

model	$I^{cross}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	0.77	0.0	1.11
<i>opt</i>	$3.33 \cdot 10^{-4}$	0.0	0.98
nn^{MLP}	$1.03 \cdot 10^{-5}$	0.0	0.97
nn^{GCN}	1.07	-0.02	0.81
nn^{SAGE}	0.56	0.0	0.52
nn^{GAT}	1.71	0.0	1.08
nn^{GATv2}	$1.89 \cdot 10^{-3}$	0.0	0.97

The prediction of nn^{MLP} provides an even better result than the one obtained from direct optimization. The directly optimized solution is closer to $u_{equiv1.01x}$ than the layers. However, this might be because the neural networks did not impose an upper bound on the SUPG parameters. The parameters predicted by the best models and the ones obtained from direct optimization are shown in Figure [15](#). The line plots in Figure [16](#) show that the predictions of both nn^{MLP} and nn^{GATv2} resolve the exponential layer well at $y = 0.5$. The function values of u_h^{GATv2} slightly decay near the lateral layers while retaining relatively sharp layers. Still, this leads to an overall worse objective function value I^{cross} .

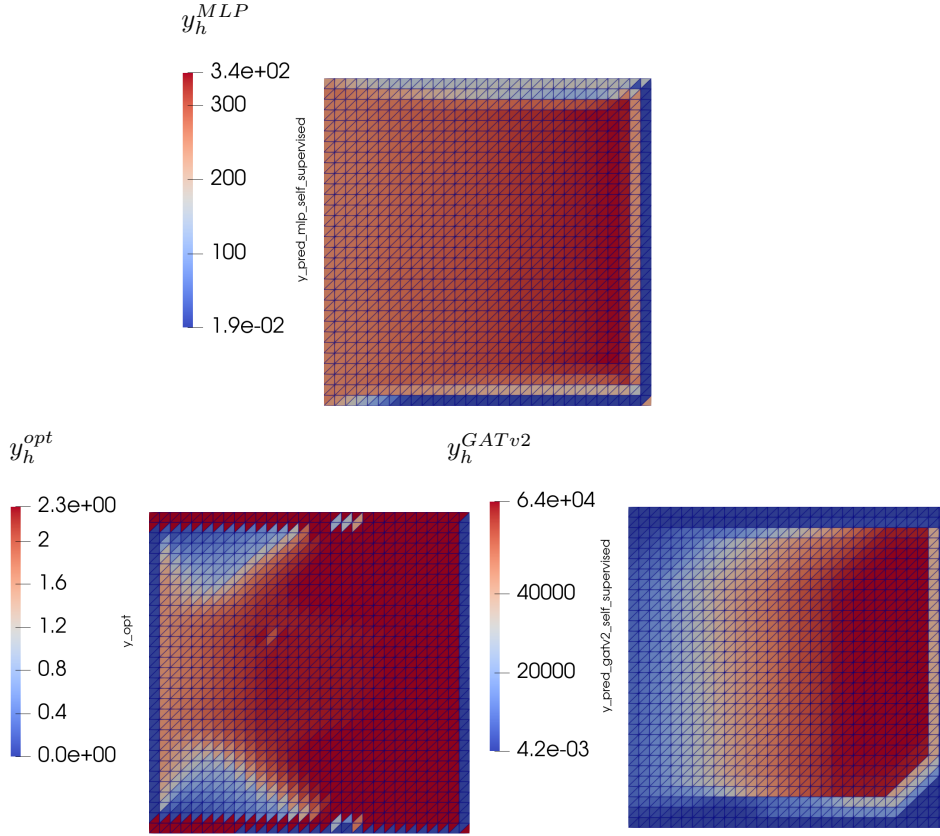


Figure 15: SPDE 1: SUPG-parameters predicted by the MLP-model, the best performing model for this problem. Parameters obtained from direct optimization and the GATv2 model are included for comparison.

None of the supervised models yielded satisfactory results in this network configuration, as each model's predictions were significantly worse than the standard solution. For the implementation on the quadrilateral mesh, the following results were obtained:

model	$I^{cross}(u_h)$	$\ y_h - y_h^{opt}\ _2^2$	$\min u_h$	$\max u_h$
nn^{MLP_s}	1.29	0.15	0.0	0.36
nn^{GCN_s}	3.64	0.1	-0.19	0.69
nn^{SAGE_s}	2.52	0.12	$-0.55 \cdot 10^{-3}$	0.64
nn^{GAT_s}	2.75	0.08	-0.12	0.61
nn^{GATv2_s}	2.28	0.1	-0.15	0.57

There is no correlation between the model's ability to fit the data, i.e., approximating y_h^{opt} , and a decreased value of the objective function I^{cross} . as shown in the table above, nn^{GAT_s} fits this data best and has the second-worst solution while nn^{MLP_s} has the worst fit and best performance. For all models, however, both the ability to fit the data and the quality of the solutions are poor, with all predicted solutions

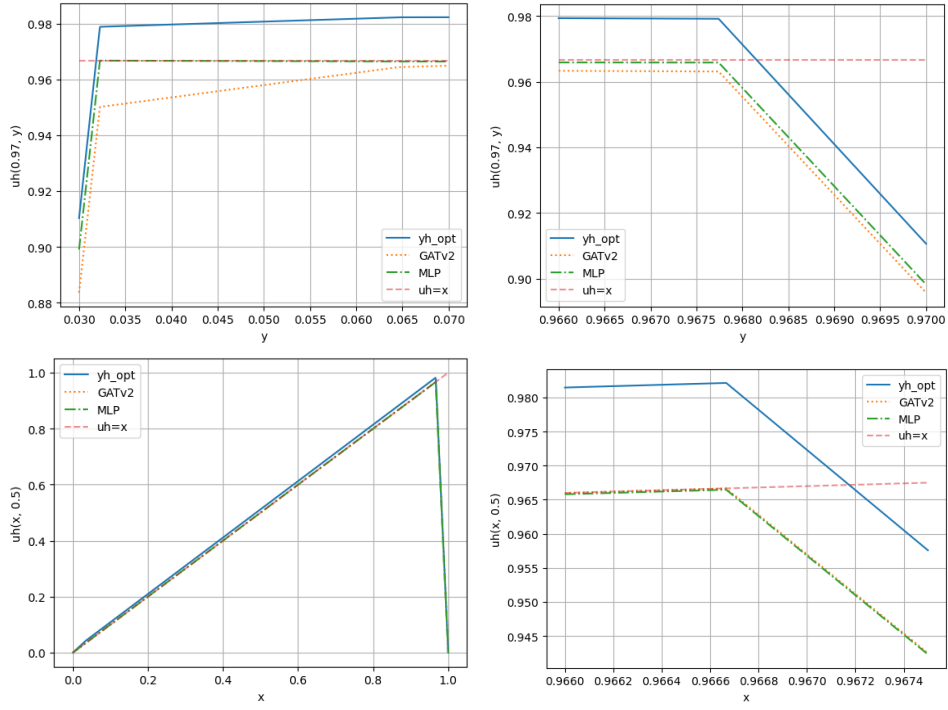


Figure 16: SPDE 1: Function values of the approximations of the MLP and GATv2 models at the exponential layer (top) $x = 1 - 1/32 \approx 0.97$ and along the streamline (bottom) at $y = 0.5$. In the bottom plots, the values for $u \equiv x$ are also shown for comparison.

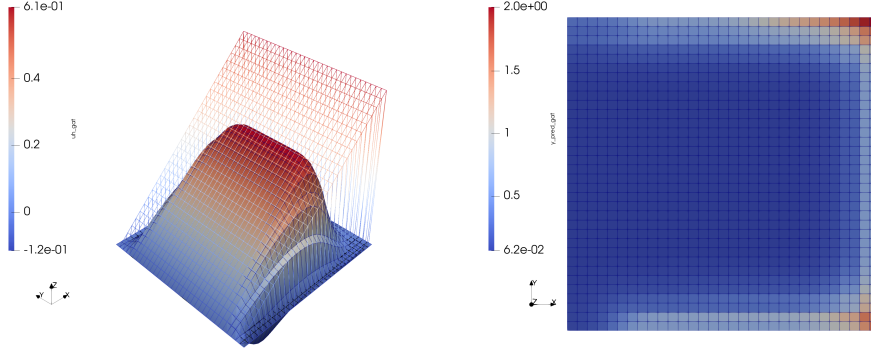


Figure 17: SPDE 1: Solution and parameters predicted by nn^{GAT_s} together with a wireframe of u_h^{opt} to show the amount of smearing.

showing considerable smearing. Figure 17 shows the solution and parameters predicted by nn^{GAT_s} together with a wireframe of the directly optimized solution to show the amount of smearing.

Possible explanations for this might be non-unique solutions and long-range dependencies of the provided solutions. As Figure 12 suggests, there are multiple possible SUPG parameters that improve on the standard solution. Restricting a model's learning to a single solution by optimizing (20) significantly limits its flexibility, a limitation that is exacerbated when the model is fit to multiple problems simultaneously. The self-supervised approach allows the model to predict a solution following the most general pattern in the training set rather than the one provided by direct optimization.

Another explanation might be that an optimal SUPG parameter for a given cell cannot be fully characterized by the local properties provided as input. The parameters provided by the direct optimization exhibit a large variation in Ω compared with the constant problem data and the standard solution and its derivatives. The optimal value might depend on all or at least more cells than the specified neighbors in the graph, especially those farther away. Figure 18 shows how a parameter change in a single cell affects the whole solution. The standard parameters were changed in one cell by 10^{-8} and then the difference

$$u_h^{std} - \tilde{u}_h^{std}$$

was computed to see how the solution changes. A change in a cell in the right lateral boundary affects all cells within the streamline, and a change in the interior of Ω has a small effect on multiple cells. Changing the parameters in the intersection of the lateral boundary $y = 1$ and the exponential layer has the largest effect. While it is clear that information is transported primarily in the upstream direction, the magnitude and dispersion are not easily predictable.

Since the supervised approach yielded poor results for all SPDEs and meshes, the results for the other problems will not be discussed to avoid redundancy.

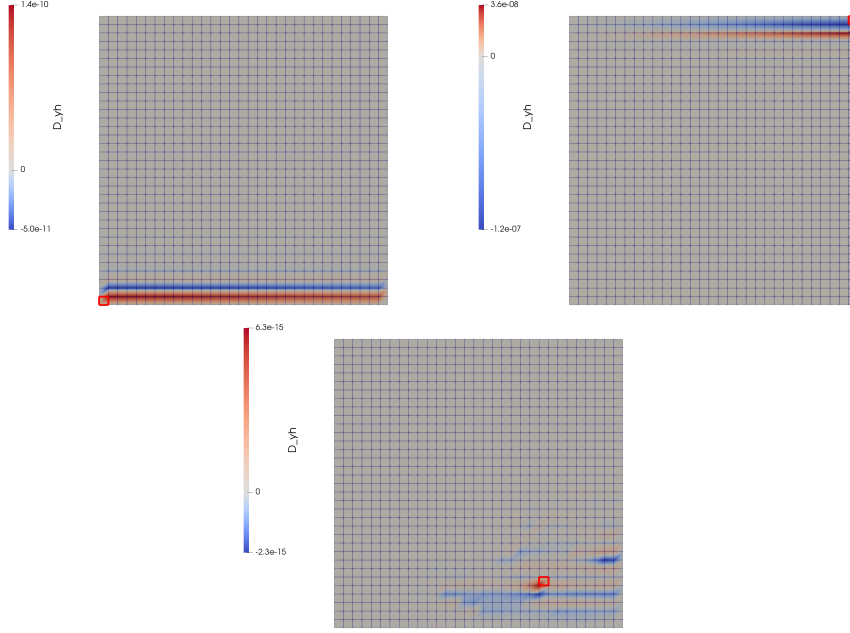


Figure 18: SPDE 1: $u_h(y_h^{std}) - u_h(y_h^{std} + e_j 10^{-8})$ where $e_j \in Y_h$ equals 1 for the j -th cell and 0 everywhere else. The cell j is indicated by a red frame.

4.2 SPDE 2

Let $\Gamma^D = \partial\Omega$, $\varepsilon = 10^{-8}$, $u_D = 0$, $\mathbf{b} = (1, 0)^T$,

$$f(x, y) = \begin{cases} 0, & \text{if } |x - 0.5| \geq 0.25 \text{ or } |y - 0.5| \geq 0.25, \\ -32(x - 0.5), & \text{otherwise.} \end{cases}$$

This problem is taken from [32] and has a discontinuous source term which is a weak derivative with respect to x of:

$$F(x, y) = \begin{cases} 0, & \text{if } |x - 0.5| \geq 0.25 \text{ or } |y - 0.5| \geq 0.25, \\ -16(x - 0.5)^2 + 1, & \text{otherwise.} \end{cases}$$

F is called a solution to the reduced problem where $\varepsilon = 0$. Figure 19 shows a plot of F and the convection field $\mathbf{b}$ scaled by f . In this example, f prescribes changes in streamline direction, unlike the previous example, where the solution's directional change was prescribed by the boundary condition. I^{cross} was used as a loss function, just as in [32].

In the test set, there are 2 implementations, one on a quadrilateral with $N_x(= 40) \times N_y(= 33)$ cells and one on a triangular mesh with $2 \times N_x(= 41) \times N_y(= 40)$ cells. The configurations of the problems used in the training set are shown in Figure 44 in the appendix.

On the quadrilateral mesh, the optimization with the L-BFGS-B algorithm terminated unsuccessfully after 220 iterations and 655 function evaluations total.

model	$I^{cross}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	0.39	-0.02	1.2
<i>opt</i>	0.24	$-0.8 \cdot 10^{-3}$	1.02
nn^{MLP}	0.31	-0.01	1.16
nn^{GCN}	0.38	-0.02	1.2
nn^{SAGE}	0.39	-0.03	1.19
nn^{GAT}	0.39	-0.02	1.19
nn^{GATv2}	0.38	-0.02	1.22

Overall, the performance of all models and direct optimization is poor, as all solutions exhibit oscillations in the lateral layers. One reason might be that n_y is not divisible by 4, so the mesh cannot resolve the source term's case statement. However, solutions to problems implemented on quadrilateral meshes in the training set exhibited similar artifacts, even when this was not the case.

On the triangular mesh, the L-BFGS-B minimization terminated unsuccessfully after 14 iterations and 138 total function evaluations.

model	$I^{cross}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	0.87	-0.13	1.11
<i>opt</i>	0.66	-0.05	0.98
nn^{MLP}	0.66	-0.03	0.99
nn^{GCN}	0.63	-0.01	0.98
nn^{SAGE}	0.63	$-5.95 \cdot 10^{-3}$	0.98
nn^{GAT}	0.79	-0.11	1.08
nn^{GATv2}	0.66	$-6.28 \cdot 10^{-4}$	0.98

While most models, except for nn^{GAT} , were able to resolve the oscillations in the lateral layers better, unphysical behavior still visibly showed in the outer part of the solution $|x - 0.5| \geq 0.25$ or $|y - 0.5| \geq 0.25$ where the solution is expected to vanish. While nn^{SAGE} has the lowest value for I^{cross} , nn^{GATv2} has the lowest oscillation in this area. Figure 20 shows the predicted solution u_h^{GATv2} , as well as the parameters y_h^{opt} , y_h^{GATv2} , y_h^{SAGE} . Since all models produce relatively similar solutions, a line plot showing important details is shown in Figure 21.

4.3 SPDE 3

Let $\Gamma^D = \partial\Omega$, $\varepsilon = 10^{-8}$, $\mathbf{b} = (\cos(-\pi/3), \sin(-\pi/3))^T$, $f = 0$,

$$u_D = \begin{cases} 0 & \text{for } x = 1 \text{ or } y \leq 0.7, \\ 1 & \text{else.} \end{cases}$$

This problem has a constant convection $\mathbf{b}$ and a vanishing source term f , so the layers are prescribed by the boundary condition. Figure 22 shows how the convection

$F(x, y)$ and $(\mathbf{b}, f(x, y))$

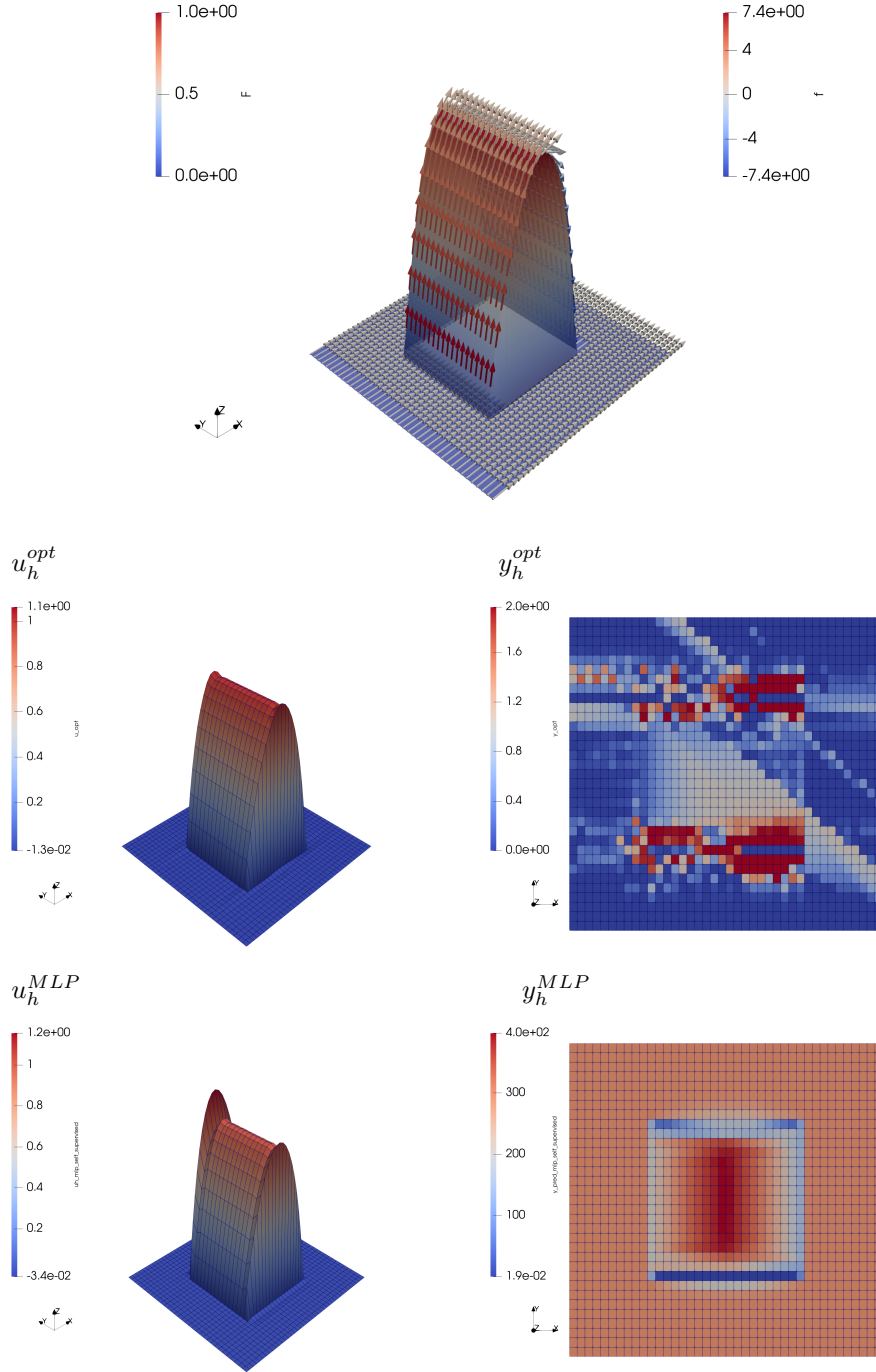


Figure 19: SPDE 2: Exact solution and SUPG parameters with solutions obtained from direct optimization and the best performing model nn^{MLP}

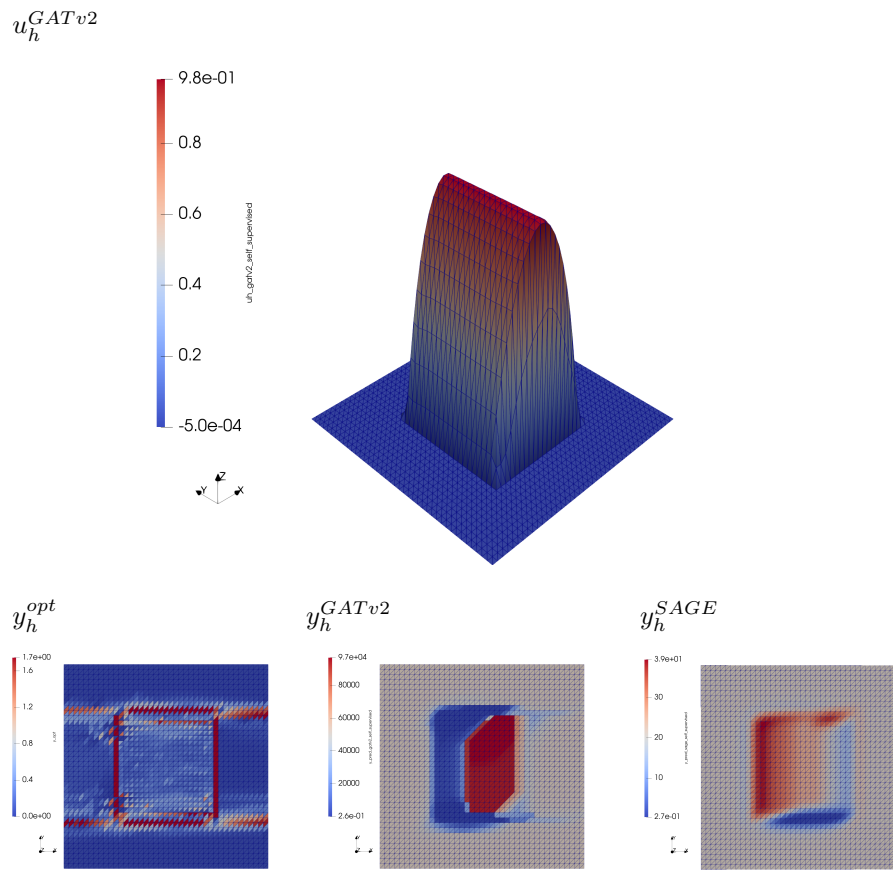


Figure 20: SPDE 2: Solution predicted by nn^{GATv2} and SUPG parameters predicted by opt , nn^{GATv2} and nn^{MLP} .

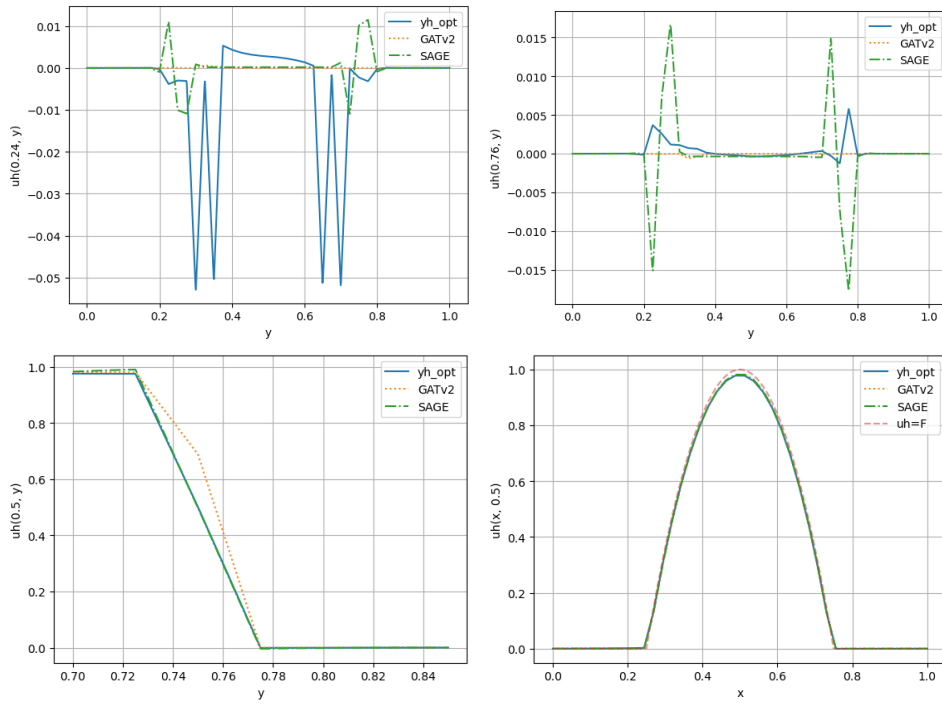


Figure 21: SPDE 2: Function values of the solutions u_h^{opt} and the best performing models nn^{GATv2} and nn^{SAGE} . The top plots show that both u_h^{opt} and u_h^{SAGE} exhibit oscillations at the layers in streamline direction $x = 10/N_x \approx 0.24$ and $x = 31/N_x \approx 0.76$ where u_h^{GATv2} is mostly smooth. This, however, is accompanied by some smearing in the lateral layers, as shown in the bottom-left plot. The bottom right plot shows the solutions at $y = 0.5$ together with F , the solution to the reduced problem.

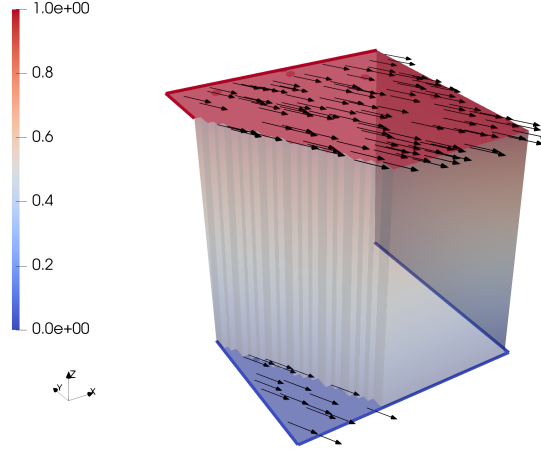


Figure 22: SPDE 3: Convection field and source term ($\mathbf{b}, f$) of SPDE 3 and reduced solution prescribed by the boundary values on the inflow boundary. The values in the outflow boundary are also indicated. The reduced solution produces a lateral layer, and the boundary values in the outflow layer produce boundary layers.

transports the boundary values from the inflow boundary $x = 0$ and $y = 1$ in Ω . Namely, the inflow boundary together with the constant source term prescribe two components for a solution to the reduced problem: $F_1 \equiv 1$ for points in the streamline direction of $(x, 1)$ and $(0, y)$, $y > 0.7$, and $F_2 \equiv 0$ for all points in the streamline direction with starting points $(0, y)$, $y \leq 0.7$. This produces a lateral layer at $y = 0.7 - \frac{\cos(-\pi/3)}{\sin(-\pi/3)}x$ and boundary layers at $x = 1$ and $(x, 0)$, $x = \frac{\sin(-\pi/3)}{\cos(-\pi/3)}0.7$.

The Q_1 test problem was implemented on an $N_x (= 42) \times N_y (= 33)$ quadrilateral mesh, and the direct optimization took 8.741 iterations and 9.348 total function evaluations. The configurations of the meshes for the training and test problems are listed in Figure 45 in the Appendix.

model	$I^{cross}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	0.58	-0.05	1.24
<i>opt</i>	0.18	-0.01	1.02
nn^{MLP}	0.29	-0.02	1.04
nn^{GCN}	0.28	-0.01	1.11
nn^{SAGE}	0.66	-0.04	1.0
nn^{GAT}	2.56	-0.03	1.06
nn^{GATv2}	0.3	-0.04	1.04

The predictions of nn^{MLP} , nn^{GCN} and nn^{GATv2} improved on the standard solution. When compared to u^{opt} , however, they all show visible smearing in the lateral layer as Figure 23 shows. Even though, it has the smallest loss value, u_h^{GCN} has a visible overshooting artifact at $x = 1$, $y = 0$ as shown in Figure 24

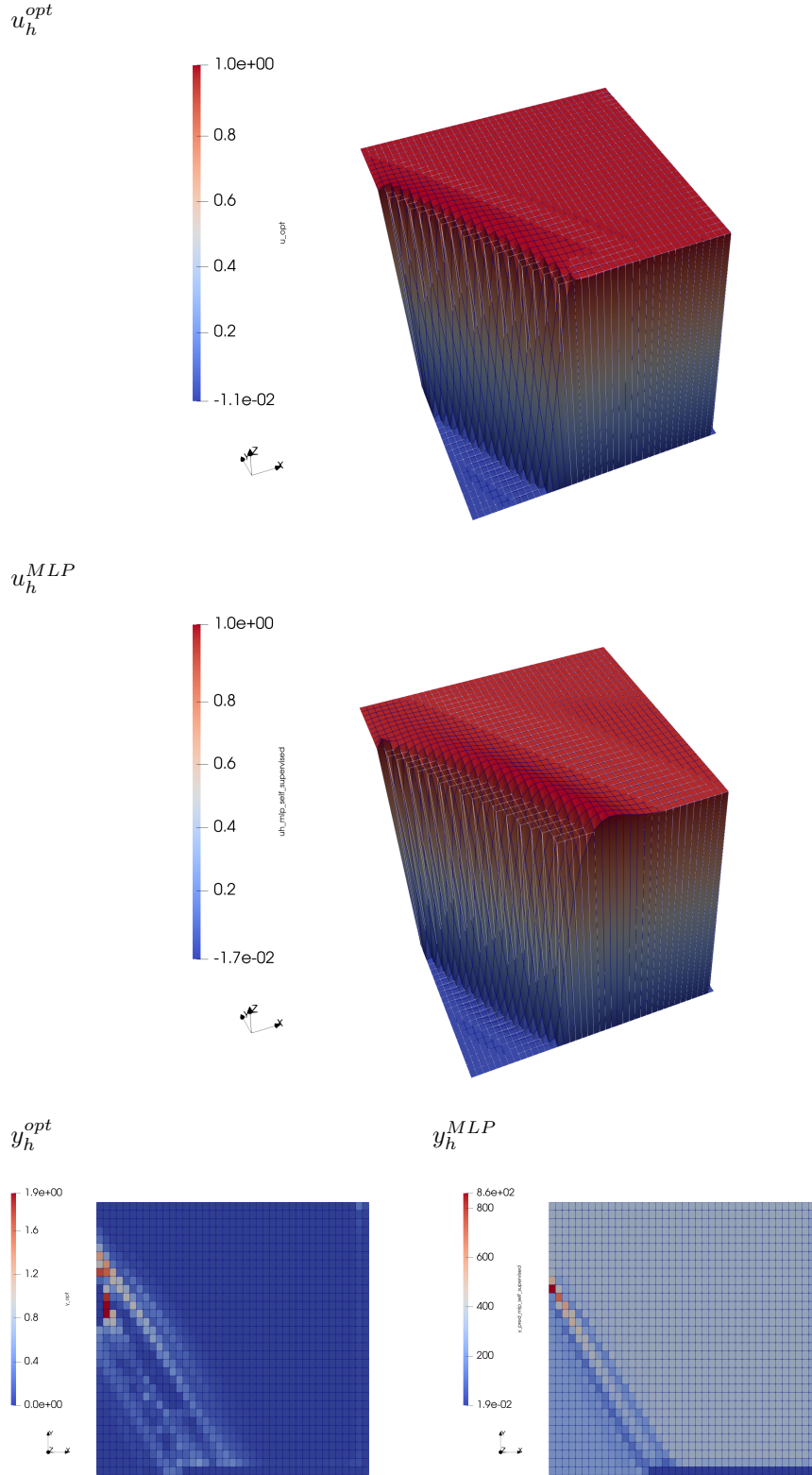


Figure 23: SPDE 3: Directly optimized parameters and approximation of the solution of SPDE 3 and prediction of nn^{MLP} on a quadrilateral mesh with $n_x = 42 \times n_y = 33$ cells. The lateral layer of the solution to the reduced problem is indicated to showcase the smearing of the predicted solution.

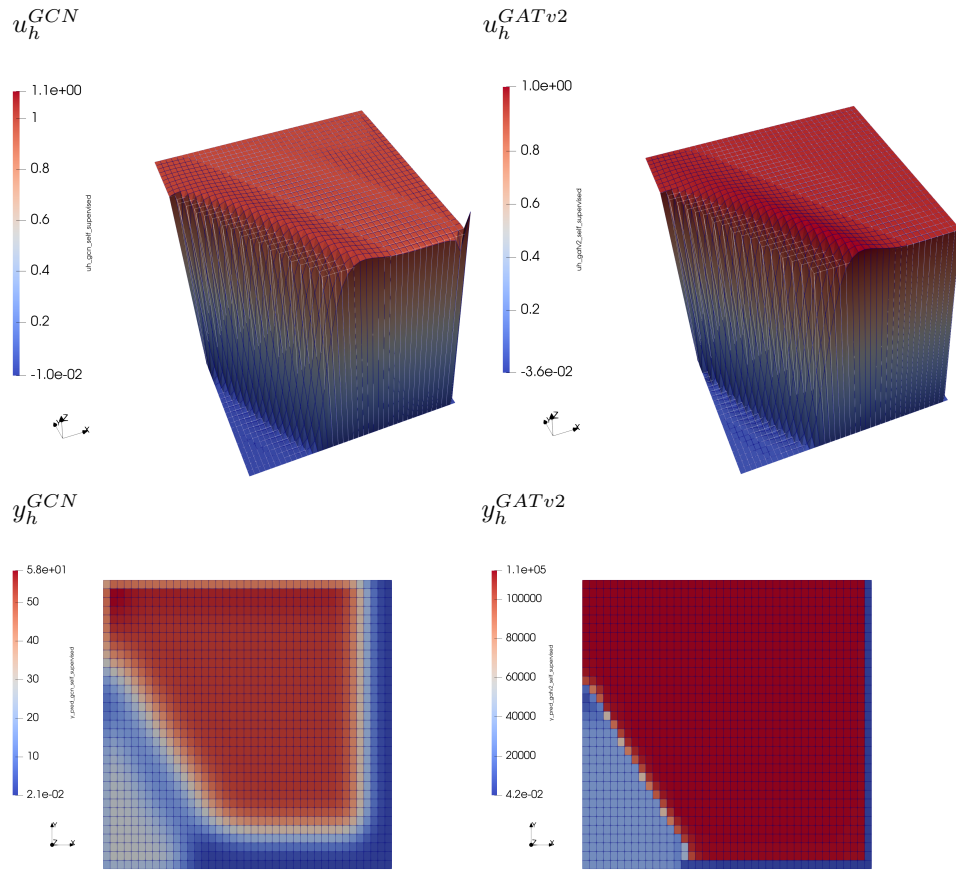


Figure 24: SPDE 3: Predictions of nn^{GCN} and nn^{GATv2} . While nn^{GCN} has the smallest loss value, nn^{GATv2} has no overshooting artifact.

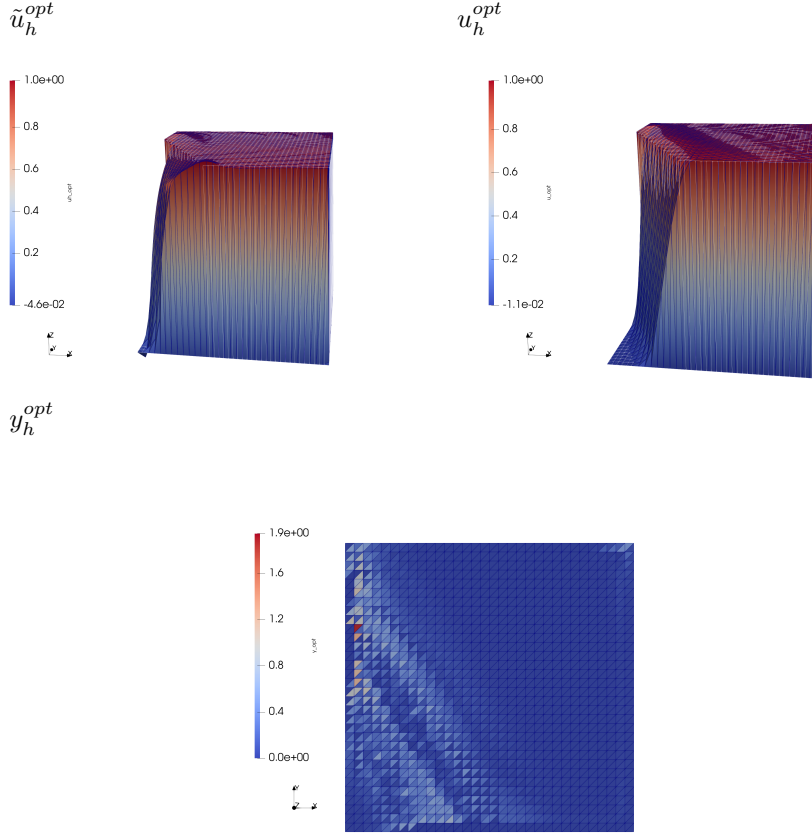


Figure 25: SPDE 3: Overoptimized approximation $\tilde{u}_h^{opt}$ and u_h^{opt} that was obtained by early stopping.

The implementation on the triangular mesh was harder to optimize. Reducing the value of I^{cross} beyond 2.3 leads to artifacts in the boundary layers, and further reducing it pushes the lateral layer towards the boundary. Consequently, the whole solution appeared to converge pointwise to 1 away from the boundary. After 1.000.000 iterations, the optimization loop was stopped, and a solution obtained after 1.062 iterations was obtained by setting $ftol$ to $6 \cdot 10^{-8}$, which was used as u_h^{opt} .

model	$I^{cross}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	0.98	-0.04	1.17
<i>opt</i>	0.23	$-4.05 \cdot 10^{-3}$	1.02
nn^{MLP}	0.4	-0.01	1.03
nn^{GCN}	0.39	$-5.23 \cdot 10^{-3}$	1.08
nn^{SAGE}	0.73	-0.03	1.02
nn^{GAT}	2.44	-0.03	1.12
nn^{GATv2}	0.79	-0.01	1.01

The only model without visible artifacts in the solution is nn^{MLP} . It still shows

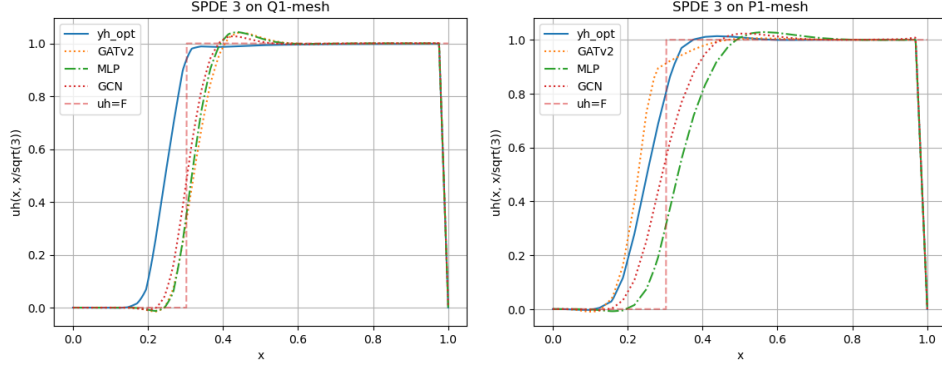


Figure 26: SPDE 3: Function values of approximations to SPDE 3 perpendicular to the lateral layer in P_1 and Q_1 finite element spaces.

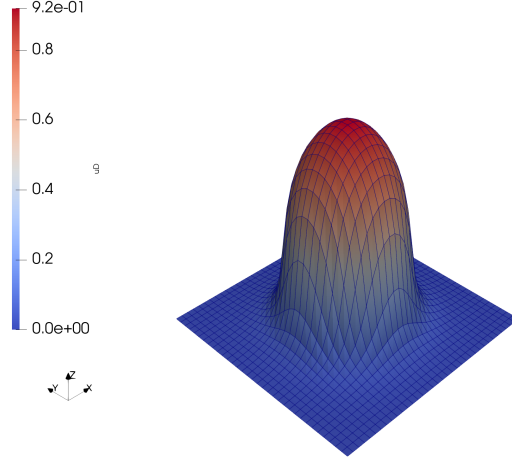


Figure 27: SPDE 4: Analytical solution.

smearing in the lateral layer, however, as Figure 25 shows. The lineplots in Figure 26 show the lateral layer at $y = \frac{-\sin(-\pi/3)}{\cos(-\pi/3)}x$ which is perpendicular to the lateral layer.

4.4 SPDE 4

Let $\Gamma^D = \partial\Omega$, $\varepsilon = 10^{-3}$, $\mathbf{b} = (2, 3)^T$ and f, u_D chosen such that

$$u = 16x(1-x)y(1-y) \left(\frac{1}{2} + \frac{\arctan[2\varepsilon^{-1/2}(0.25^2 - (x-0.5)^2 - (y-0.5)^2)]}{\pi} \right),$$

which was already described in Section 1. This problem is only mildly convection dominated, since smaller ε values lead to larger quadrature errors as demonstrated in [16]. As test problems a Q_1 -approximation on a $N_x (= 35) \times N_y (= 33)$ quadrilateral mesh and a P_1 -approximation on a $2 \times N_x (= 33) \times N_y (= 39)$ mesh were implemented. The configurations for the training problems are shown in Figure 45 in the Appendix. The solution to this problem is shown in Figure 27. Since an analytical solution is readily available for this problem, I^{L^2} was used as a loss function.

Direct optimization can reduce the L^2 -error to $5.3 \cdot 10^{-7}$ after 5.000 iterations. The result can be further improved with more iterations.

model	$I^{L^2}(u_h)$	$\min u_h$	$\max u_h$
<i>opt</i>	$5.3 \cdot 10^{-7}$	0.0	0.92
<i>std</i>	$1.17 \cdot 10^{-5}$	$-2.6 \cdot 10^{-3}$	0.92
<i>nn^{MLP}</i>	$8.75 \cdot 10^{-3}$	$-1.87 \cdot 10^{-3}$	0.92
<i>nn^{GCN}</i>	$5.91 \cdot 10^{-3}$	$-5.97 \cdot 10^{-4}$	0.92
<i>nn^{SAGE}</i>	$5.81 \cdot 10^{-3}$	$-8.69 \cdot 10^{-4}$	0.92
<i>nn^{GAT}</i>	$4.64 \cdot 10^{-3}$	-0.01	0.92
<i>nn^{GATv2}</i>	$8.11 \cdot 10^{-3}$	$-1.88 \cdot 10^{-3}$	0.92

None of the models improves on the standard result. In fact, all of them have yielded worse results than the standard solution. The same applies to the implementation on the triangular mesh. This might be because the standard solution already yields a small error, which is dominated by the other loss values in the training set.

model	$I^{L^2}(u_h)$	$\min u_h$	$\max u_h$
<i>opt</i>	$5.84 \cdot 10^{-9}$	0	0.92
<i>std</i>	$8.35 \cdot 10^{-6}$	$-1.14 \cdot 10^{-3}$	0.92
<i>nn^{MLP}</i>	$3.18 \cdot 10^{-4}$	-0.01	0.92
<i>nn^{GCN}</i>	$2.17 \cdot 10^{-4}$	$-5.23 \cdot 10^{-3}$	0.92
<i>nn^{SAGE}</i>	$2.21 \cdot 10^{-4}$	-0.03	0.92
<i>nn^{GAT}</i>	$6.21 \cdot 10^{-3}$	-0.03	0.92
<i>nn^{GATv2}</i>	$5.73 \cdot 10^{-4}$	-0.01	0.92

Even though all models perform worse than the standard solution, they are close enough to the optimal solution as Figure 28 shows.

4.5 SPDE 5

Let $\Gamma^D = \partial\Omega$, $\varepsilon = 10^{-3}$, $\mathbf{b} = (2, 3)^T$, $c = 1$ and f, u_D chosen so that:

$$u = xy^2 - y^2 \exp\left(\frac{2(x-1)}{\varepsilon}\right) - x \exp\left(\frac{3(y-1)}{\varepsilon}\right) + \exp\left(\frac{2(x-1) + 3(y-1)}{\varepsilon}\right).$$

The SPDE was implemented on 2 test meshes: a quadrilateral one with $N_x (= 32) \times N_y (= 37)$ cells and a triangular one with $2 \times N_x (= 33) \times N_y (= 35)$ cells. The solution u is shown in Figure 29. In contrast to the previous problem, however, the standard parameters do not provide a satisfactory result as u_h^{std} exhibits an overshooting artifact in the boundary layer at $x = 1$.

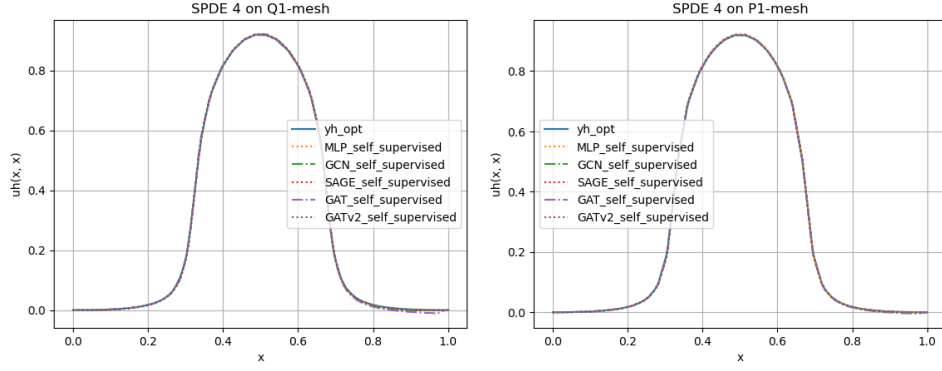


Figure 28: SPDE 4: Approximations of the self-supervised models and the analytical solution as u_h^{opt} on both test problems.

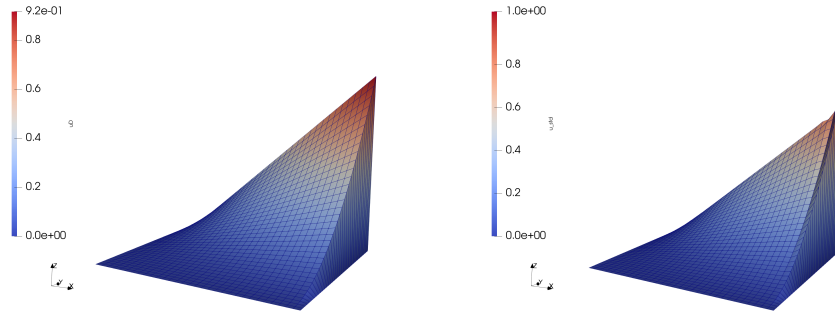


Figure 29: SPDE 5: Exact solution and approximation with standard SUPG parameters in Q_1 on a quadrilateral 32×37 mesh.

model	$I^{L^2}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	$1.67 \cdot 10^{-4}$	0	1.05
<i>opt</i>	$1.19 \cdot 10^{-9}$	0	0.92
nn^{MLP}	$2.57 \cdot 10^{-4}$	-0.02	0.91
nn^{GCN}	$6.2 \cdot 10^{-4}$	$-1.11 \cdot 10^{-3}$	0.88
nn^{SAGE}	$2.82 \cdot 10^{-2}$	$-6.14 \cdot 10^{-2}$	0.39
nn^{GAT}	$8,34 \cdot 10^{-4}$	-0.77	0.77
nn^{GATv2}	$1.19 \cdot 10^{-2}$	$-8.20 \cdot 10^{-3}$	0.68

For both test problems, all models provided worse results than the standard approximation. Figure 30 shows the prediction of nn^{GCN} which has only a marginally smaller overshooting artifact at $x = y = 1$ and exhibits smearing along the layer at $x = 1$ and $y = 1$. As with the previous problem, the relatively small loss values, even for qualitatively bad solutions, cause the 20 implementations of SPDE 5 in the training set have little influence on the model's output. The exact configurations are described in Figure 47 in the Appendix. On the triangular mesh, a direct optimization with 5.000 iterations yielded an $L^2 - error$ of $3.83 \cdot 10^{-9}$.

model	$I^{L^2}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	$1.02 \cdot 10^{-4}$	0	1.02
<i>opt</i>	$3.83 \cdot 10^{-9}$	0.0	0.92
nn^{MLP}	$4.81 \cdot 10^{-4}$	$-9.68 \cdot 10^{-3}$	0.91
nn^{GCN}	$3.2 \cdot 10^{-4}$	$-4.84 \cdot 10^{-4}$	0.87
nn^{SAGE}	$2.73 \cdot 10^{-2}$	-0.06	0.4
nn^{GAT}	$1.09 \cdot 10^{-3}$	0.0	0.73
nn^{GATv2}	$1.21 \cdot 10^{-2}$	-0.02	0.72

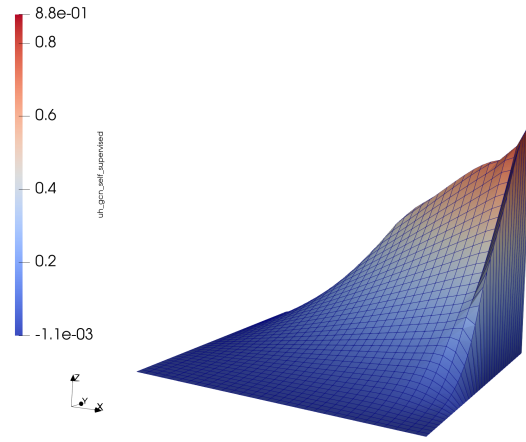
4.6 SPDE 6

Let $\Gamma^N = \{0\} \times (0, 1)$, $\Gamma^D = \partial\Omega \setminus \Gamma^N$, $\varepsilon = 10^{-8}$, $\mathbf{b} = (-y, x)^T$, $f = g = 0$,

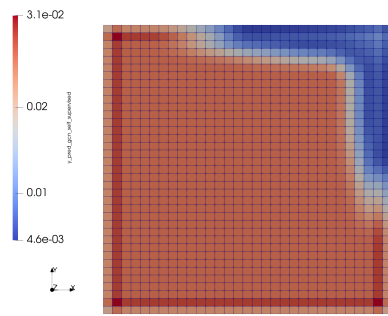
$$u_D(x, 0) = \begin{cases} x, & \text{if } 0 \leq x \leq \frac{1}{3}, \\ \frac{1}{3} + x, & \text{if } \frac{1}{3} < x < \frac{2}{3}, \\ 1 - x, & \text{if } \frac{2}{3} \leq x \leq 1, \end{cases}$$

and $u_D = 0$ elsewhere on Γ^D . In this example, the convection runs in a circular direction around the origin, so that for some point $(x_0, 0)$, $x_0 \in (0, 1)$, all points (x, y) with $r := \sqrt{x^2 + y^2}$ are in the streamline direction. The prescribed values at the inflow boundary are shown in Figure 33. As the source term is zero and the Neumann boundary condition at the outflow boundary also vanishes, a reduced solution transports the Dirichlet boundary values on $(x, 0)$ unchanged along the

u_h^{GCN}



y_h^{GCN}



y_h^{opt}

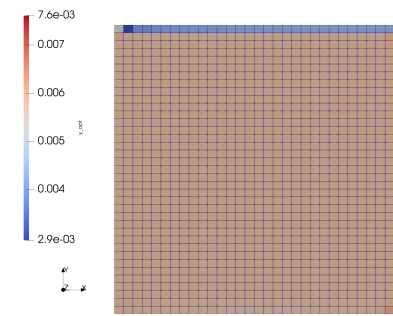


Figure 30: SPDE 5: Approximation of nn^{GCN} and parameter obtained from nn^{GCN} and direct optimization.

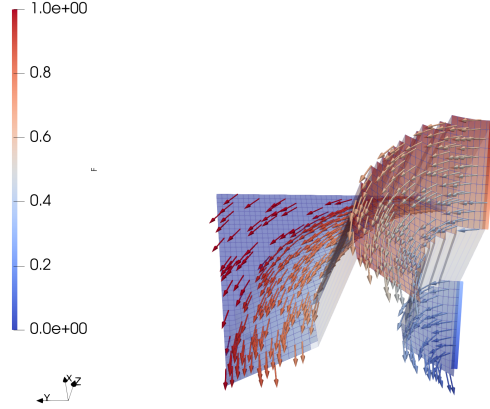


Figure 31: SPDE 6: Solution to the reduced problem with the Dirichlet boundary values at the inflow boundary and the convection field $\mathbf{b}$ indicated.

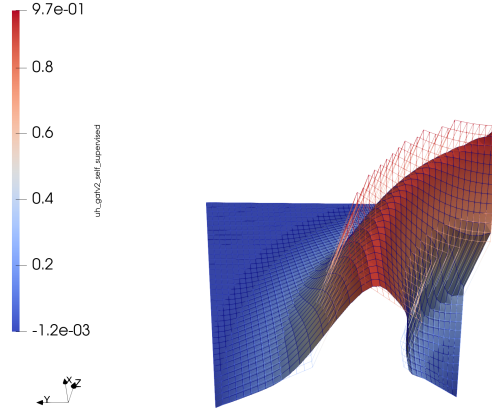
streamline as shown in Figure 31. This produces two lateral layers at $r = 1/3$ and $r = 2/3$.

This example is taken from [20], as is the used loss function I^{lim} (14) with $t_0 = 0.03$. In the test set, there are two implementations of this problem: one on a $N_x (= 39) \times N_y (= 39)$ quadrilateral mesh and one on a $2 \times N_x (= 34) \times N_y (= 30)$ triangular mesh. On the quadrilateral mesh, direct optimization required 237 iterations and 315 function evaluations to meet the $ftol$ stopping criterion. The configurations of the test and training problems are shown in Figure 48 in the Appendix.

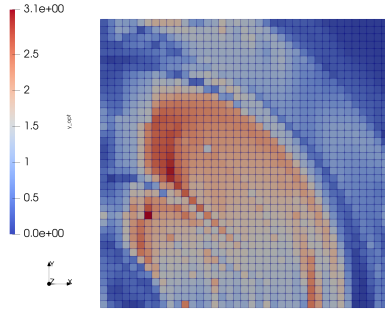
model	$I^{lim}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	$2.81 \cdot 10^{-2}$	$-1.97 \cdot 10^{-3}$	0.99
<i>opt</i>	$7.77 \cdot 10^{-3}$	$1.17 \cdot 10^{-3}$	0.97
nn^{MLP}	$1.01 \cdot 10^{-2}$	$-1.12 \cdot 10^{-3}$	0.97
nn^{GCN}	$7.78 \cdot 10^{-3}$	$-1.16 \cdot 10^{-3}$	0.97
nn^{SAGE}	$8.02 \cdot 10^{-3}$	$-1.14 \cdot 10^{-3}$	0.97
nn^{GAT}	$1.21 \cdot 10^{-2}$	$-1.81 \cdot 10^{-3}$	0.98
nn^{GATv2}	$7.95 \cdot 10^{-3}$	$-1.16 \cdot 10^{-3}$	0.97

All models improved on the standard solution, and nn^{GCN} and nn^{GATv2} perform almost as well as the directly optimized approximation. All approximations, however, did exhibit smearing in both lateral layers, like for example in the plot of u_h^{GATv2} in Figure 32. A closer examination of the predictions suggests that u_h^{GAT} yields the best results, as it is closest to the solution of the reduced problem even though it has the highest loss value. The lineplots in Figure 33 show that the standard solution is even closer, indicating that the specific loss function is not ideal for this mesh configuration. For the P_1 test problem, direct optimization took 134 iterations and 191 total function evaluations.

u_h^{GATv2}



y_h^{opt}



y_h^{GATv2}

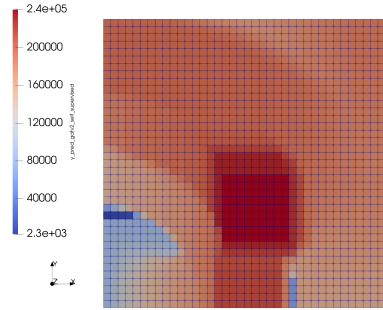


Figure 32: SPDE 6: Prediction for the approximation from nn^{GATv2} and parameters obtained from direct optimization.

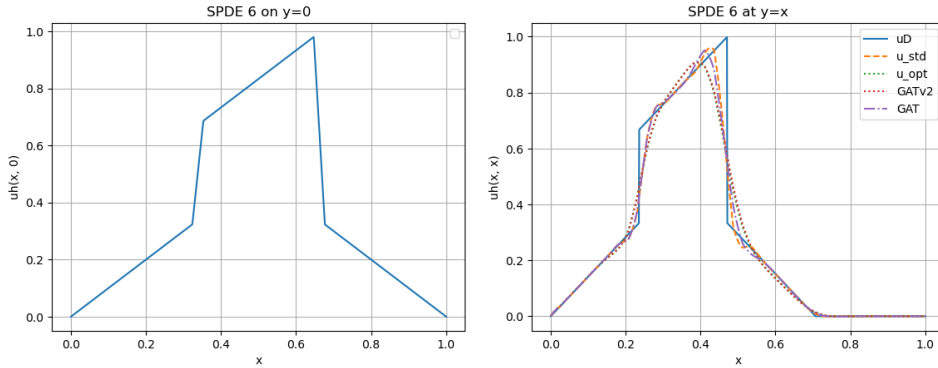


Figure 33: SPDE 6: Function values of the boundary condition at the inflow boundary $y = 0$ and the approximations of the self-supervised models along $x = y$ for the implementation on the quadrilateral mesh. The green line representing the values of u_{opt} is barely visible because it is very close to the red line that represents the prediction of nn^{GATv2} .

model	$I^{lim}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	$5.40 \cdot 10^{-2}$	$-3.37 \cdot 10^{-3}$	0.98
<i>opt</i>	$1.93 \cdot 10^{-2}$	$-1.73 \cdot 10^{-3}$	0.98
nn^{MLP}	$2.06 \cdot 10^{-2}$	$-1.82 \cdot 10^{-3}$	0.98
nn^{GCN}	$1.95 \cdot 10^{-2}$	$-1.82 \cdot 10^{-3}$	0.98
nn^{SAGE}	$1.97 \cdot 10^{-2}$	$-1.77 \cdot 10^{-3}$	0.98
nn^{GAT}	$3.05 \cdot 10^{-2}$	$-3.07 \cdot 10^{-3}$	0.98
nn^{GATv2}	$1.97 \cdot 10^{-2}$	$-1.85 \cdot 10^{-3}$	0.98

4.7 SPDE 7

Let $\Gamma^N = \{0\} \times (0, 1)$, $\Gamma^D = \partial\Omega \setminus \Gamma^N$, $\varepsilon = 10^{-8}$, $\mathbf{b} = (-y, x)^T$, $f = g = 0$,

$$u_D(x, 0) = \begin{cases} 6x, & \text{if } 0 \leq x \leq \frac{1}{6}, \\ 2 - 6x, & \text{if } \frac{1}{6} \leq x \leq \frac{1}{3}, \\ -2 + 6x, & \text{if } \frac{1}{3} \leq x \leq \frac{1}{2}, \\ 4 - 6x, & \text{if } \frac{1}{2} \leq x \leq \frac{2}{3}, \\ -4 + 6x, & \text{if } \frac{2}{3} \leq x \leq \frac{5}{6}, \\ 6 - 6x, & \text{if } \frac{5}{6} \leq x \leq 1, \end{cases}$$

and $u_D = 0$ elsewhere on Γ^D . This problem is also taken from [20] and was optimized using I^{lim} with $t_0 = 1$. It differs from the previous problem only in the definition of the boundary values at the inflow boundary.

In the test set, two problems were implemented: One on a $N_x (= 31) \times N_y (= 42)$ quadrilateral mesh and one on a $2 \times N_x (= 31) \times N_y (= 42)$ triangular mesh. The configurations of the training problems are shown in Figure 49 in the Appendix. Since computing the loss I^{lim} was more costly, and optimization took a very long time, the stopping criteria for the direct optimization were changed to $\text{ftol} = 10^{-11}$. The optimization on the quadrilateral mesh took 6.374 iterations and 6.854 function evaluations.

model	$I^{lim}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	$3.32 \cdot 10^{-2}$	$-1.1 \cdot 10^{-2}$	0.97
<i>opt</i>	$2.1 \cdot 10^{-2}$	$7.22 \cdot 10^{-3}$	0.97
nn^{MLP}	$2.1 \cdot 10^{-2}$	$-7.77 \cdot 10^{-3}$	0.97
nn^{GCN}	$2.32 \cdot 10^{-2}$	$-7.21 \cdot 10^{-3}$	0.97
nn^{SAGE}	$2.11 \cdot 10^{-2}$	$-7.42 \cdot 10^{-3}$	0.97
nn^{GAT}	$2.81 \cdot 10^{-2}$	$-116 \cdot 10^{-2}$	0.97
nn^{GATv2}	$2.19 \cdot 10^{-2}$	$-8.07 \cdot 10^{-3}$	0.97

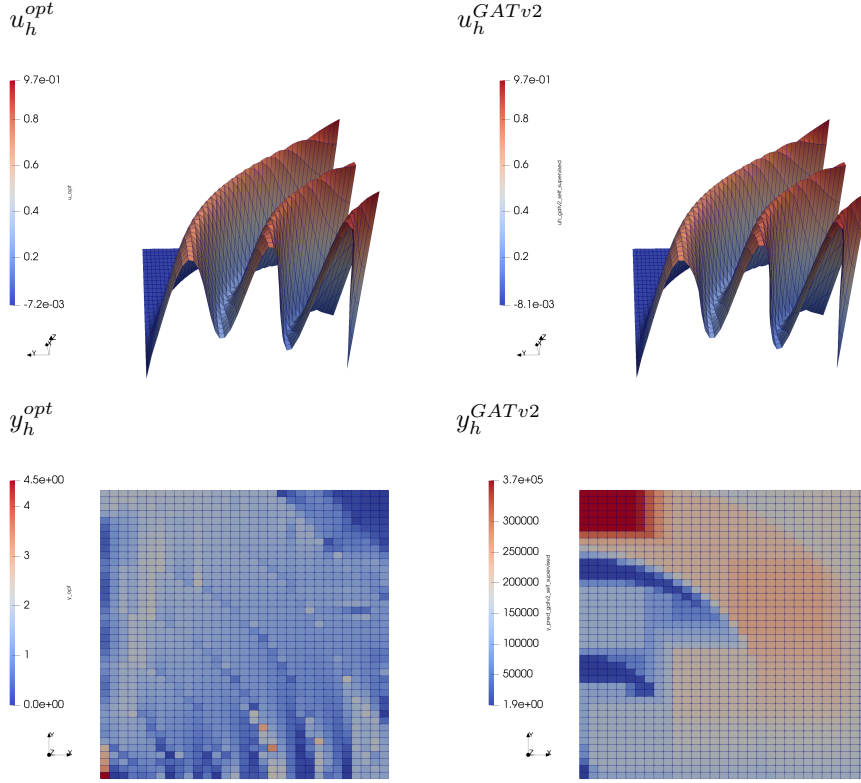


Figure 34: SPDE 7: Directly optimized solution and prediction of nn^{GATv2} on a quadrilateral mesh.

Overall, the results are comparable to the results for the previous SPDE, only that all predictions exhibit even more smearing of the solution at $x = y$ as shown in Figure 35. Again, although it has the highest loss value, the prediction from nn^{GAT} is closest to the solution of the reduced problem, with only the standard solution yielding even better results. Figure 34 shows a comparison between u_h^{opt} and u_h^{GATv2} . It can be seen that nn^{GATv2} approximates a solution close to the directly optimized solution.

The direct optimization on the triangular mesh took 1.704 iterations and 1.756 function evaluations.

model	$I^{lim}(u_h)$	$\min u_h$	$\max u_h$
<i>std</i>	0.14	$-1.9 \cdot 10^{-2}$	0.97
<i>opt</i>	$5.57 \cdot 10^{-2}$	$-7.7 \cdot 10^{-3}$	0.97
nn^{MLP}	$5.59 \cdot 10^{-2}$	$-8.05 \cdot 10^{-3}$	0.97
nn^{GCN}	$5.9 \cdot 10^{-2}$	$-7.64 \cdot 10^{-3}$	0.97
nn^{SAGE}	$5.64 \cdot 10^{-2}$	$-7.96 \cdot 10^{-3}$	0.97
nn^{GAT}	$1.12 \cdot 10^{-1}$	$-2.03 \cdot 10^{-2}$	0.97
nn^{GATv2}	$5.7 \cdot 10^{-2}$	$-9.23 \cdot 10^{-3}$	0.97

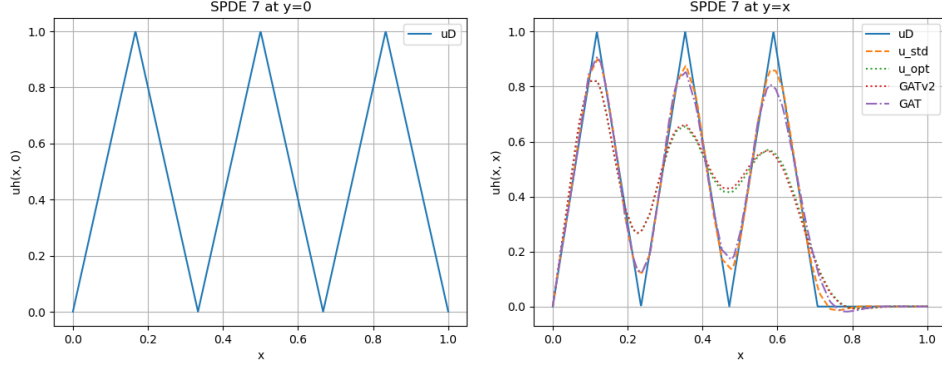


Figure 35: SPDE 7: Dirichlet boundary conditions at the inflow boundary and function values of the model predictions on the quadrilateral mesh along $x = y$.

SPDE	MLP	GCN	$SAGE$	GAT	$GATv2$
1	best P_1 , Q_T	P_1 , Q_T	P_1 , Q_T	P_1 , Q_T	best Q_1
2	P_1 , Q_T	P_1 , Q_T	P_1 , Q_T	P_1 , Q_T	best P_1 , Q_T
3	best Q_1 , best P_1	P_1 , Q_T	P_1 , Q_T	P_1 , Q_T	P_1
4		best P_1		best Q_1	
5	P_1 , Q_T	P_1 , Q_T	P_1 , Q_T	P_1 , Q_T	P_1 , Q_T
6	best P_1 , best Q_1				
7	best P_1 , best Q_1				

Figure 36: Overview table: If a predicted solution in the test set has visible artifacts or unphysical behavior and is thus unusable, it is indicated by the corresponding function space being striked out. The best results are also indicated.

4.8 Discussion and outlook

Although the results of the presented models are not optimal, the preceding discussions indicate that incorporating data from adjacent cells can be beneficial. Many convolutional architectures can process finite-element data, with GATv2 being the most promising to date. The table in Figure 36 shows which model performed best and indicates unusable results for a certain SPDE. The overall best results are still produced by the local MLP approach, even though it also yields more unusable results than nn^{GATv2} . In comparison, nn^{GATv2} yielded only unusable results for problems where all other models also failed, with SPDE 2 and SPDE 3 as the only exceptions.

In theory, all the graph models discussed have greater expressive capacity than the MLP model, since they apply a linear transformation to the input nodes. The GATv2 architecture has been shown to be more expressive than GAT, and is assumed here to be the most expressive of the architectures discussed. Increasing the expressivity of the local MLP approach yields better predictions for some problems but worse performance for others if it is not increased enough. This indicates that GCN, SAGE, and GAT are harder to fit but are also more prone to overfitting. Only

GATv2 has enough expressivity to overcome this tradeoff. Additionally, GATv2 has fewer unusable results than all others. It can be concluded that GATv2 has the best generalization.

The suboptimal results may be explained by the dataset’s heterogeneity and thus might be improved with more data. This data is difficult to obtain because results must be judged qualitatively, and loss functions do not always reliably predict unphysical behavior, as shown in section 1. Another problem was the high variance in loss values across different problems and even within the same problem when implemented on different meshes. This leads to the gradients of some problems dominating the parameter updates, causing poor generalization. A desirable trait for a loss function is that it reflects a degree of quality across all problems in the training set.

One way to provide qualitatively good data more reliably would be to produce it using a better algorithm than direct optimization, like, for example, algebraically stabilized methods as presented in 19.

Another approach might be to define adjacency not in terms of cell adjacency, but rather in terms of other graph structures for the same nodes. For example, a cell can be connected to all cells in the streamline direction. Alternatively, dense attention layers without masking can be used to predict a suitable graph structure. Transformer architectures that rely solely on dense attention layers might also improve results.

5 Revisiting the supervised approach

In [23], Petr Knobloch and Manoj Prakash obtained better results with supervised learning than the ones discussed here. As of June 2026, this paper has been submitted but has not yet been published. It was shared in a private conversation and is quoted with the authors' permission.

This section discusses only those aspects of their work that successfully addressed problems that arose during the design and training of the previously discussed models. To show the effectiveness of these approaches, two revised models were trained on a single example problem. Namely, a local MLP, which is similar to a model described by Knobloch and Prakash, and a graph neural network based on the GATv2 architecture for comparison. It will be shown that the GATv2 architecture can also be beneficial in this revised setting.

5.1 Issues encountered in the previous approaches

The preceding discussion indicated that the initial implementation was not sufficiently developed to achieve satisfactory performance. Among other things, the following aspects posed significant problems.

5.1.1 Definition of an appropriate loss functional

Several training problems could not be trained with optimal parameters because the underlying optimal control problem was not well-suited to eliminating spurious oscillations or preserving sharp layers. For example, the directly optimized solution of SPDE 2 on the triangular test mesh still showed some spurious oscillations at the lateral layers, as shown in Figure 21, and the one on the quadrilateral test mesh even had an overshooting artifact, as shown in Figure 19. The Problems SPDE 6 and SPDE 7 both exhibited smearing in the lateral layers, and the implementation of SPDE 3 on the triangular test mesh had to be stopped early to prevent the layer from being pushed towards the boundary, as shown in Figure 25.

To circumvent this issue, the authors of [23] made use of a more powerful approximation algorithm to compute an optimal approximation of the solution u . This approximation was obtained via an algebraic stabilized method, namely algebraic flux correction with BJK limiter. This method was presented in [4], and approximations obtained from this method preserve some desirable properties like the discrete maximum principle, which will not be discussed further as it is beyond the scope of this work.

We call the approximation from the AFC method $u_h^{AFC_{BJK}}$. The loss function used to compute optimal SUPG-parameters can then be defined as:

$$I^{AFC_{BJK}}(u_h) = \|u_h - u_h^{AFC_{BJK}}\|_{L^2(\Omega)} \quad (27)$$

for all problems in the training set. This is a much more robust loss functional. In addition, the design of $I^{AFC_{BJK}}$ alleviates the problem of different loss functionals yielding loss values of different magnitudes, which can disproportionately affect training.

Similar to I^{lim} , the BJK limiter has a user-chosen constant. However, in most cases, it can simply be set to 1 and still yield satisfactory results. This was done in the implementation of the revised models presented in this section, which simplifies

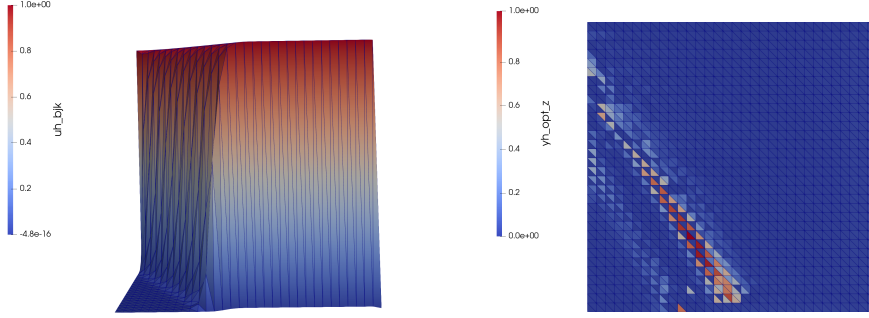


Figure 37: SPDE 3: AFC approximation with BJK limiter of SPDE 3 on a triangular mesh (left) and SUPG parameters obtained by minimizing $I^{AFC_{BJK}}(u_h)$ (right).

the optimization as optimal SUPG parameters for all problems can be obtained by minimizing the same functional. Figure 37 shows an implementation of SPDE 3 on a triangular mesh with $N_x (= 32) \times N_y (= 32)$ cells, where the triangles were obtained by dividing a square cell from the top left to bottom right. This is the opposite orientation used in the training set of the previous approach, chosen because it aligns the mesh better with the lateral layer. The SUPG parameters shown were minimized by applying a box constraint with a lower bound of 0 and an upper bound of 100 times y_h^{std} , and using scipy's minimize function with $gtol = ftol = 10^{-12}$. The corresponding solution has a loss value of $I^{AFC_{BJK}}(u_h(yh^{opt})) \approx 1,6817 \cdot 10^{-6}$.

5.1.2 Scope of the model

The previously discussed models were trained on triangular and quadrilateral meshes with varying aspect ratios that did not necessarily align with layers. The authors of [23] trained dedicated models for triangular meshes with regular aspect ratios for triangular anisotropic meshes that had been obtained by adaptive refinement at the layers. This highlights the difficulty in training models that can handle a variety of mesh structures.

It was already discussed that multiple parameters can provide satisfactory results. Deviations from the target values that are less extreme than those predicted by nn^{GATv2} can be obtained by choosing different starting values for the direct optimization or by setting different constraints. Figure 38 shows SUPG parameters for SPDE 3 obtained by direct optimization with different starting values and constraints. The top-left image shows the target parameters with $I^{AFC_{BJK}}(u_h^{opt}) = 1,6817 \cdot 10^{-6}$ obtained with 0 as the starting value. The top-right image was obtained by using y^{std} as the starting value. The loss value is $I^{AFC_{BJK}}(u_h) = 2,3148 \cdot 10^{-6}$. The bottom-left image was obtained by setting an upper constraint of $5 \cdot y_h^{std}$ instead of $100 \cdot y_h^{std}$. The loss value is $I^{AFC_{BJK}}(u_h) = 3,0780 \cdot 10^{-6}$. The bottom-right image shows the same parameters as in the upper-left image, except that values at nodes far from the layers and the outflow boundary were increased by 0.01. The cells with perturbed values are indicated by a green outline. The loss value is $I^{AFC_{BJK}}(u_h) = 2.227 \cdot 10^{-4}$.

These results indicate a drawback of the supervised approach: the model lacks the flexibility to learn the most general pattern rather than the one prescribed by the target values. To assess the level of precision a network needs to provide good results, we can perturb the assigned target values and analyze the effect on $I^{AFC_{BJK}}$.

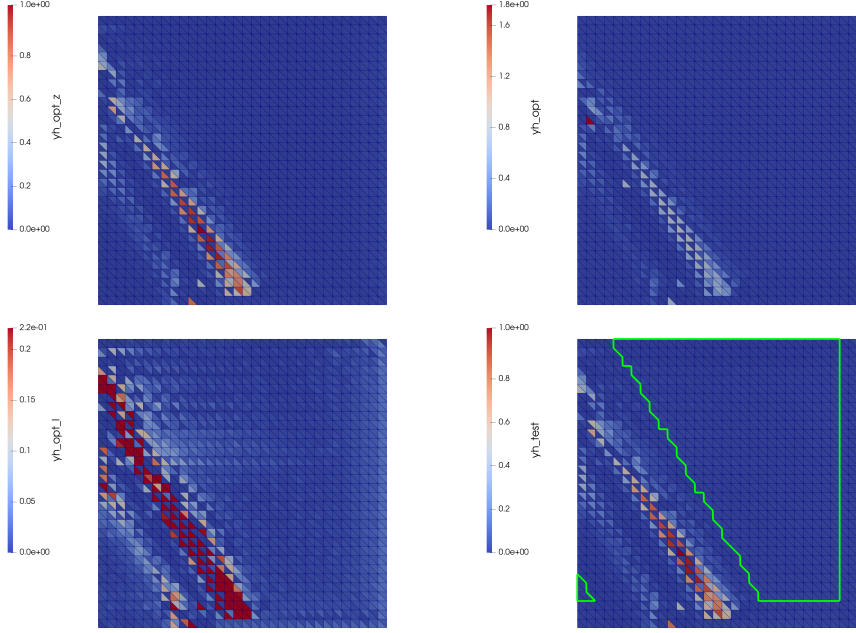


Figure 38: SPDE 3: SUPG parameters obtained by direct optimization with different starting values and constraints.

Figure 39 shows the optimal solution perturbed by normally distributed noise of increasing magnitude against the loss value $I^{AFC_{BJK}}$. It is clear that values with an l^2 -distance less than or equal to 10^{-3} can be expected to yield satisfactory results, but that there are also results farther away that can give much better results than randomly perturbed results. It also shows that it cannot be concluded that nodes away from the layers and the outflow boundary are unimportant to the quality of the solution.

A theoretical lower bound on the precision of the local MLP approach for each input problem can be obtained by analyzing which values in the input tensor coincide. The best local predictor for cells with identical input values is the mean of all their target values. For the example discussed here, this is $\|y_h^{pred} - y_h^{op}\|_{l^2} = 2,349 \cdot 10^{-2}$ with the given input values. For a graph neural network, all input values of the cells within the perceptive field of the node have to coincide. For the example discussed here, this poses no lower bound on precision. This shows that the GATv2 architecture can also improve results using the more refined approach in 23.

In 23, the SUPG parameters for an input problem were computed by three different models: one for interior cells, one for boundary cells at the inflow, and one for boundary cells at the outflow. Each model has differently designed input parameters to account for specific properties of those cells. These input parameters will not be discussed here, as improvements over the previous results can already be obtained without reproducing them. Restricting the network to interior and boundary nodes yields a lower, albeit not much lower, bound on the model's precision. In the example discussed here, a new bound is $2,339 \cdot 10^{-2}$. As it turns out, training dedicated models for each cell type still yields noticeable improvements.

The revised models presented here only predict the SUPG parameters for the interior cells. The results in the cited paper did not yield optimal results in the exterior

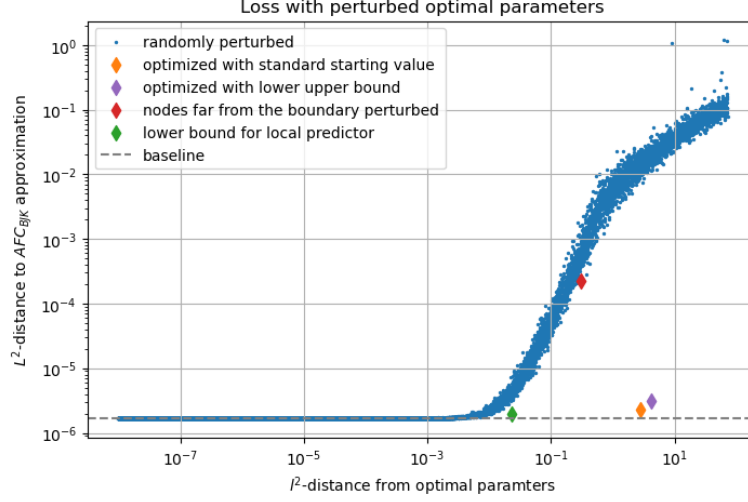


Figure 39: Optimal solution perturbed by normally distributed noise of increasing magnitude against the loss value $I^{AFC_{BJK}}$ together with the approximations obtained from the parameters shown in Figure 38.

cells in the outflow direction and needed additional postprocessing to suppress spurious oscillations. The SUPG parameters used to compute $I^{AFC_{BJK}}$ were obtained by setting the predicted values for interior nodes and the target values y_h^{opt} for boundary nodes.

5.1.3 Model size and architecture

The models from Section 4 were comparably small with a maximal layer width of 10. Knobloch and Prakash used much wider networks with widths of up to 256 nodes. The network for unstructured meshes concatenated linear layers of widths

$$3 - 256 - 256 - 64 - 256 - 256 - 64 - 256 - 256 - 128 - 1$$

with different non-linear activation functions. Namely, ReLU in most layers and in the $256 - 64 - 256$ blocks, a Sigmoid, and a SiLU function, which is just the input multiplied by the value of the Sigmoid function of the input.

The last block of shape $256 - 128 - 1$ appeared twice, causing the model to produce two separate outputs from two separate two-layer perceptrons that share the same 256-dimensional input from the previous layers. As with the other aspects, there were more complex architectural decisions that are not discussed here, as the ones presented already improved the revised models.

For the revised MLP model, the architecture described above was copied, and a GATv2 model with 4 layers, each with a channel size of 256 and ReLU activation functions, was implemented. This GATv2 model produced a 256-dimensional output, which was then processed by two two-layer perceptrons in the same way as the MLP.

The models in 23 used fewer input dimensions that aggregate data. For example $\|\nabla u_h^{std}\|$ evaluated at the barycenter is used instead of $\partial_x u_h^{std}$ and $\partial_y u_h^{std}$. The revised models implemented here used the same parameters defined in Section 4.

Only the last parameter, indicating whether the mesh is triangular or quadrilateral, is omitted because we restrict the model’s scope to triangular meshes.

5.1.4 Optimization loop

Instead of choosing an Adam optimizer and reducing the learning rate whenever the optimization stagnates, a cosine annealing schedule was implemented, which reduces the learning rate over T_i epochs after each epoch according to

$$lr_{t+1} = \frac{0.01}{2} \left(1 + \cos \left(\frac{T_{cur}}{T_i} \pi \right) \right), \quad (28)$$

where T_{cur} is a running index like t which is set to 0 whenever it is equal to T_i . The initial value for T_{cur} is set to 50. After T_i epochs, the value T_i is doubled. This optimization loop allows the model to escape shallow local minima by repeatedly restarting the learning rate at a higher value, with increasingly long phases between restarts as training progresses.

Instead of an l^2 -error, the models computed a Huber loss:

$$l_1(y_h) = \sum_{i=1}^{dim Y_h} \begin{cases} 0.5((y_h)_i - (y_h^{opt})_i)^2 & |(y_h)_i - (y_h^{opt})_i| \leq 1 \\ |(y_h)_i - (y_h^{opt})_i| - 0.5 & \text{else.} \end{cases}$$

and a non-normalized l^2 -loss:

$$l_2(y_h) = \sum_{i=1}^{dim Y_h} ((y_h)_i - (y_h^{opt})_i)^2.$$

The final loss was computed by:

$$l(y_h) = 0.7 l_1(y_h) + 0.3 l_2(y_h). \quad (29)$$

It is not specified that summation was used instead of averaging. However, this is not a significant difference, as the latter is simply a rescaling of both loss values by the same factor, and the models automatically rescale large gradients to a norm of 1.

5.2 Results of the revised models

Figure 40 shows the loss values of the training loop for the problem from Figure 37 for both models. It can be seen that the GATv2 model is better at fitting the data and at producing approximations closer to $u_h^{AFC_{BJK}}$. The MLP model was stopped after 200.000 epochs, and the GATv2 model after 150.000 epochs, since the GATv2 model performed better but was more computationally expensive. For each model, the final version with the lowest loss value of $I^{AFC_{BJK}}$ was chosen. The plot at the bottom is a zoomed-in version of Figure 39, with the loss values for the MLP and the GATv2 prediction added for comparison. It can be seen that none of the models reach the theoretical lower bound for a local predictor. The GATv2 prediction, however, performs noticeably better than the MLP prediction and the approximation obtained by direct optimization with a tighter upper bound on y_h . The table in Figure 41 shows the results of the revised models.

Figure 42 shows that both predictions exhibit unphysical behavior in the outflow boundary, even though the values in those cells were chosen to be optimal. This can be explained by the interior values not being optimal and by long-range dependencies, as discussed in Section 4.1. The bottom-right image shows a line plot

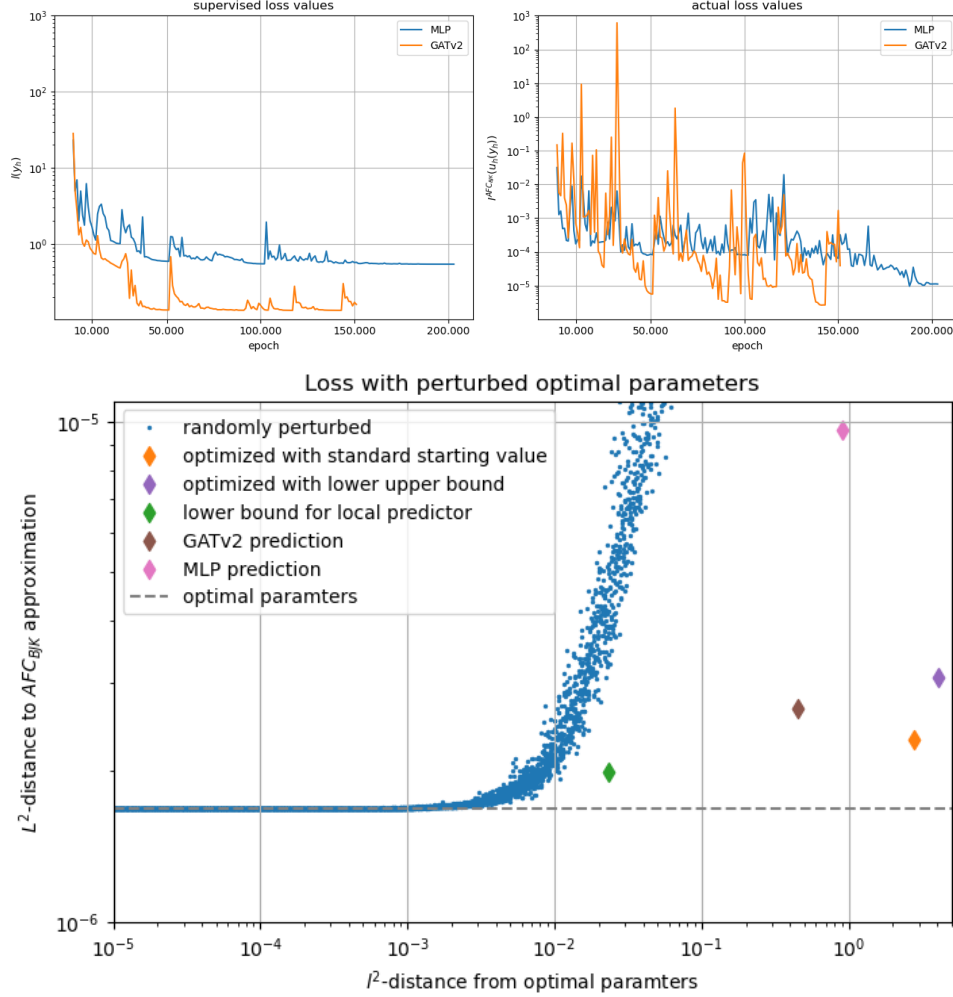


Figure 40: Supervised loss (29) and loss values of I^{AFC}_{BJK} (27) for the revised MLP and GATv2 model. For comparison, the I^{AFC}_{BJK} -, and the l^2 -error of the MLP and the GATv2 prediction are shown in a zoomed-in view of Figure 39.

model	$\max(u_h)$	$\min(u_h)$	$l(y_h)$	$\ y_h - y_h^{AFC_{BJK}}\ _{l^2}$	$I^{AFC_{BJK}}$
revised MLP	1.04	-0.0006	0.546	0.9153	$9.643 \cdot 10^{-6}$
revised GATv2	1.01	-0.0008	0.1346	0.4551	$2.66 \cdot 10^{-6}$

Figure 41: Results of the revised models.

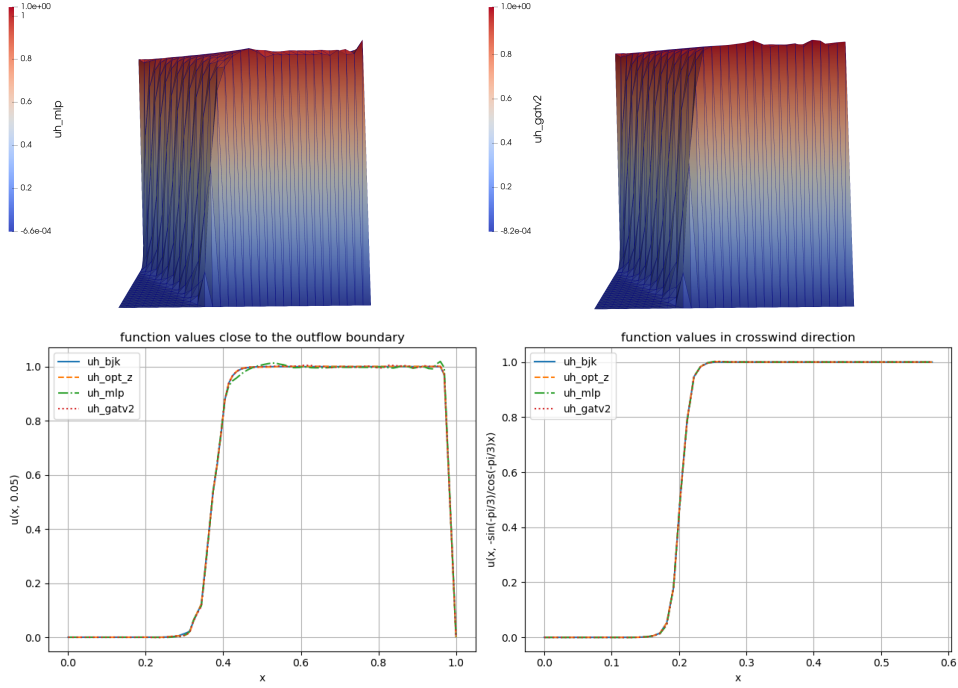


Figure 42: SPDE 3: Approximations obtained from the predictions of the revised MLP model (left) and the revised GATv2 model (right).

of the function values at the outflow boundary of the predictions of both models, the directly optimized solution and the AFC_{BJK} -approximation. It can be seen that the unphysical behavior is most pronounced for the MLP prediction. The predicted function values for the lateral layer away from the outflow boundary are close to optimal for both predictions, as shown in the line plot in the bottom-left image, which shows the function values in crosswind direction starting from the origin.

It can be concluded that the GATv2 architecture has more desirable theoretical properties, which are reflected in the results of this experiment. Even though none of the models reached the theoretical lower bound for a local predictor, the GATv2 outperformed the MLP both in fitting the data and in approximating the solution of the AFC scheme with the BJK limiter. When comparing the predictions of the supervised models shown in Figure 42 with the predictions of the self-supervised models shown in Figure 26, it can be concluded that the results presented here are more accurate. It should, however, be noted that the self-supervised models were trained on a much more heterogeneous dataset and were designed using a less developed approach and much smaller models. In the simpler setup of the previous models, the supervised approach failed, which indicates that the self-supervised approach has the potential to be utilized to train more robust and flexible models

6 Appendix

Code is hosted in a public GitHub repository at [https://github.com/vschormann/](https://github.com/vschormann/MA_ML_assisted_SUPG)

[MA_ML_assisted_SUPG](#)

Nr.	mesh type	n_x	n_y
test_spde.000	q	32	34
test_spde.001	t	30	32
spde.000	q	8	8
spde.001	q	8	16
spde.002	q	8	32
spde.003	q	8	64
spde.004	q	16	8
spde.005	q	16	16
spde.006	q	16	32
spde.007	q	16	64
spde.008	q	32	8
spde.009	q	32	16
spde.010	q	32	32
spde.011	q	64	8
spde.012	q	64	16
spde.013	q	64	32
spde.054	t	16	16
spde.055	t	16	32
spde.056	t	32	16
spde.057	t	32	32

Figure 43: implementations of SPDE 1

Nr.	mesh type	n_x	n_y
test_spde.002	q	40	33
test_spde.003	t	41	40
spde_014	q	8	16
spde_015	q	8	32
spde_016	q	8	64
spde_017	q	16	16
spde_018	q	32	16
spde_019	q	32	32
spde_020	q	32	64
spde_021	q	64	32
spde_022	q	64	64
spde_058	p	16	16
spde_059	p	16	32
spde_060	p	32	16
spde_061	p	32	32

Figure 44: implementations of SPDE 2

Nr.	mesh type	n_x	n_y
test_spde.008	q	41	33
test_spde.009	t	32	32
spde_070	q	16	32
spde_071	q	32	16
spde_072	q	32	32
spde_081	t	16	32
spde_082	t	32	16
spde_083	t	32	32

Figure 45: implementations of SPDE 3

Nr.	mesh type	n_x	n_y
test_spde.006	q	35	33
test_spde.007	t	33	39
spde_039	q	8	8
spde_040	q	8	16
spde_041	q	8	32
spde_042	q	8	32
spde_043	q	16	8
spde_044	q	16	16
spde_045	q	16	32
spde_046	q	16	64
spde_047	q	32	16
spde_048	q	32	32
spde_049	q	32	64
spde_050	q	64	8
spde_051	q	64	16
spde_052	q	64	32
spde_053	q	64	64
spde_066	t	16	16
spde_067	t	16	32
spde_068	t	32	16
spde_069	t	32	32

Figure 46: implementations of SPDE 4

Nr.	mesh type	n_x	n_y
test_spde.004	q	32	37
test_spde.005	t	33	35
spde.023	q	8	8
spde.024	q	8	16
spde.025	q	8	32
spde.026	q	8	64
spde.027	q	16	8
spde.028	q	16	16
spde.029	q	16	32
spde.030	q	16	64
spde.031	q	32	8
spde.032	q	32	16
spde.033	q	32	32
spde.034	q	32	64
spde.035	q	64	8
spde.036	q	64	16
spde.037	q	64	32
spde.038	q	64	64
spde.062	t	16	16
spde.063	t	16	32
spde.064	t	32	16
spde.065	t	32	32

Figure 47: implementations of SPDE 5

Nr.	mesh type	n_x	n_y
test_spde.010	q	39	39
test_spde.011	t	34	30
spde_073	q	16	16
spde_074	q	16	32
spde_075	q	32	16
spde_076	q	32	32
spde_084	t	16	16
spde_085	t	16	32
spde_086	t	32	16
spde_087	t	32	32

Figure 48: implementations of SPDE 6

Nr.	mesh type	n_x	n_y
test_spde.012	q	31	42
test_spde.013	t	31	42
spde_077	q	16	16
spde_078	q	16	32
spde_079	q	32	16
spde_080	q	32	32

Figure 49: implementations of SPDE 7

Sources

- [1] Martin S. Alnaes et al. “The FEniCS Project Version 1.5”. In: *Archive of Numerical Software* 3 (2015). DOI: [10.11588/ans.2015.100.20553](https://doi.org/10.11588/ans.2015.100.20553).
- [2] Martin S. Alnaes et al. “Unified Form Language: A domain-specific language for weak formulations of partial differential equations”. In: *ACM Transactions on Mathematical Software* 40 (2014). DOI: [10.1145/2566630](https://doi.org/10.1145/2566630).
- [3] Igor A. Baratta et al. *DOLFINx: the next generation FEniCS problem solving environment*. preprint. 2023. DOI: [10.5281/zenodo.10447666](https://doi.org/10.5281/zenodo.10447666).
- [4] Gabriel R. Barrenechea, Volker John, and Petr Knobloch. *Finite element methods respecting the discrete maximum principle for convection-diffusion equations*. 2023. arXiv: [2204.07480 \[math.NA\]](https://arxiv.org/abs/2204.07480). URL: <https://arxiv.org/abs/2204.07480>.
- [5] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations and Concepts*. Springer, 2024.
- [6] Shaked Brody, Uri Alon, and Eran Yahav. *How Attentive are Graph Attention Networks?* 2022. arXiv: [2105.14491 \[cs.LG\]](https://arxiv.org/abs/2105.14491). URL: <https://arxiv.org/abs/2105.14491>.
- [7] Richard H. Byrd et al. “A Limited Memory Algorithm for Bound Constrained Optimization”. In: *SIAM Journal on Scientific Computing* 16.5 (1995), pp. 1190–1208. DOI: [10.1137/0916069](https://doi.org/10.1137/0916069). eprint: <https://doi.org/10.1137/0916069>. URL: <https://doi.org/10.1137/0916069>.
- [8] Hanqiu Chen et al. *Bottleneck Analysis of Dynamic Graph Neural Network Inference on CPU and GPU*. 2023. arXiv: [2210.03900 \[cs.AR\]](https://arxiv.org/abs/2210.03900). URL: <https://arxiv.org/abs/2210.03900>.
- [9] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: <http://jmlr.org/papers/v12/duchi11a.html>.
- [10] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. arXiv: [1903.02428 \[cs.LG\]](https://arxiv.org/abs/1903.02428). URL: <https://arxiv.org/abs/1903.02428>.
- [11] François. *The Little Book of Deep Learning*. Writers Republic LLC, 2023. ISBN: 9789732346495. URL: <https://books.google.de/books?id=dZsFOAEACAAJ>.
- [12] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2018. arXiv: [1706.02216 \[cs.SI\]](https://arxiv.org/abs/1706.02216). URL: <https://arxiv.org/abs/1706.02216>.
- [13] Isaac Harari and Thomas J.R. Hughes. “What are C and h?: Inequalities for the analysis and design of finite element methods”. In: *Computer Methods in Applied Mechanics and Engineering* 97.2 (1992), pp. 157–192. ISSN: 0045-7825. DOI: [https://doi.org/10.1016/0045-7825\(92\)90162-D](https://doi.org/10.1016/0045-7825(92)90162-D). URL: <https://www.sciencedirect.com/science/article/pii/004578259290162D>.
- [14] Thomas Joseph Robert Hughes and Alexander Nelson Brooks. “MULTI-DIMENSIONAL UPWIND SCHEME WITH NO CROSSWIND DIFFUSION.” In: *AMD*, vol. 34. ASME (1979).
- [15] Volker John and Petr Knobloch. “On spurious oscillations at layers diminishing (SOLD) methods for convection–diffusion equations: Part I – A review”. In: *Computer Methods in Applied Mechanics and Engineering* 196 (2007), pp. 2197–2215.

- [16] Volker John, Petr Knobloch, and Simona B. Savescu. “A posteriori optimization of parameters in stabilized methods for convection–diffusion problems – Part I”. In: *Computer Methods in Applied Mechanics and Engineering* 200.41 (2011), pp. 2916–2929. ISSN: 0045-7825. DOI: <https://doi.org/10.1016/j.cma.2011.04.016>. URL: <https://www.sciencedirect.com/science/article/pii/S0045782511001496>.
- [17] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014). URL: <https://api.semanticscholar.org/CorpusID:6628106>.
- [18] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: [1609.02907 \[cs.LG\]](https://arxiv.org/abs/1609.02907). URL: <https://arxiv.org/abs/1609.02907>.
- [19] Petr Knobloch. *An algebraically stabilized method for convection-diffusion-reaction problems with optimal experimental convergence rates on general meshes*. 2022. arXiv: [2208.07705 \[math.NA\]](https://arxiv.org/abs/2208.07705). URL: <https://arxiv.org/abs/2208.07705>.
- [20] Petr Knobloch, Petr Lukáš, and Pavel Solin. “On error indicators for optimizing parameters in stabilized methods”. In: *Advances in Computational Mathematics* 45.4 (2019), pp. 1853–1862. DOI: [10.1007/s10444-019-09662-4](https://doi.org/10.1007/s10444-019-09662-4). URL: <https://doi.org/10.1007/s10444-019-09662-4>.
- [21] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. DOI: [10.1007/978-3-642-23099-8](https://doi.org/10.1007/978-3-642-23099-8).
- [22] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *CoRR* abs/1912.01703 (2019). arXiv: [1912.01703](https://arxiv.org/abs/1912.01703). URL: <http://arxiv.org/abs/1912.01703>.
- [23] Manoj Prakash Petr Knobloch. *Application of machine learning to the computation of parameters in the SUPG method for unstructured and anisotropic meshes*. 2026.
- [24] Matthew W. Scroggs et al. “Basix: a runtime finite element basis evaluation library”. In: *Journal of Open Source Software* 7.73 (2022), p. 3982. DOI: [10.21105/joss.03982](https://doi.org/10.21105/joss.03982).
- [25] Matthew W. Scroggs et al. “Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes”. In: *ACM Transactions on Mathematical Software* 48.2 (2022), 18:1–18:23. DOI: [10.1145/3524456](https://doi.org/10.1145/3524456).
- [26] Roos H.G. and M. Stynes and L. Tobiska. *Robust Numerical Methods for Singularly Perturbed Differential Equations: Convection-Diffusion-Reaction and Flow Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. DOI: [10.1007/978-3-540-34467-4_1](https://doi.org/10.1007/978-3-540-34467-4_1).
- [27] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: [1710.10903 \[stat.ML\]](https://arxiv.org/abs/1710.10903). URL: <https://arxiv.org/abs/1710.10903>.
- [28] R. Verfürth. “Robust A Posteriori Error Estimates for Stationary Convection-Diffusion Equations”. In: *SIAM Journal on Numerical Analysis* 43.4 (2005), pp. 1766–1782. DOI: [10.1137/040604261](https://doi.org/10.1137/040604261). eprint: <https://doi.org/10.1137/040604261>. URL: <https://doi.org/10.1137/040604261>.
- [29] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: [10.1038/s41592-019-0686-2](https://doi.org/10.1038/s41592-019-0686-2).

- [30] Han Xiao, Kashif Rasul, and Roland Vollgraf. “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms”. In: *CoRR* abs/1708.07747 (2017). arXiv: [1708.07747](https://arxiv.org/abs/1708.07747). URL: <http://arxiv.org/abs/1708.07747>.
- [31] Sangeeta Yadav and Sashikumaar Ganesan. “Artificial neural network-augmented stabilized finite element method”. In: *Journal of Computational Physics* 499 (2024), p. 112702. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2023.112702>. URL: <https://www.sciencedirect.com/science/article/pii/S0021999123007970>.
- [32] Sangeeta Yadav and Sashikumaar Ganesan. “ConvStabNet: a CNN-based approach for the prediction of local stabilization parameter for SUPG scheme”. In: *Calcolo* 61.3 (Aug. 2024). ISSN: 0008-0624. DOI: [10.1007/s10092-024-00597-x](https://doi.org/10.1007/s10092-024-00597-x). URL: <https://doi.org/10.1007/s10092-024-00597-x>.
- [33] Sangeeta Yadav and Sashikumaar Ganesan. “SPDE-Net: Neural Network based prediction of stabilization parameter for SUPG technique”. In: *Proceedings of The 13th Asian Conference on Machine Learning*. Ed. by Vineeth N. Balasubramanian and Ivor Tsang. Vol. 157. Proceedings of Machine Learning Research. PMLR, 17–19 Nov 2021, pp. 268–283. URL: <https://proceedings.mlr.press/v157/yadav21a.html>.

Fachbereich Mathematik, Informatik und Physik

SELBSTSTÄNDIGKEITSERKLÄRUNG

Name: Schormann	(BITTE nur Block- oder Maschinenschrift verwenden.)
Vorname(n): Paul Otto Vincent	
Studiengang: M. Sc. Mathematik	
Matr. Nr.: 5097352	

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Datum: 11.07.2026

Unterschrift: 