Reinforcement Learning for Adaptive Mesh Refinement

Jiachen Yang[*] LLNL

Tarik Dzanic[*] Texas A&M University

Brenden Petersen[*] LLNL

Jun Kudo[*] LLNL

Ketan Mittal LLNL

Vladimir Tomov LLNL

Jean-Sylvain Camier LLNL

Tuo Zhao Georgia Tech

Hongyuan Zha Georgia Tech

Tzanio Kolev LLNL

Robert Anderson LLNL

Daniel Faissol LLNL

Abstract

Finite element simulations of physical systems governed by partial differential equations (PDE) crucially depend on adaptive mesh refinement (AMR) to allocate computational budget to regions where higher resolution is required. Existing scalable AMR methods make heuristic refinement decisions based on instantaneous error estimation and thus do not aim for long-term optimality over an entire simulation. We propose a novel formulation of AMR as a Markov decision process and apply deep reinforcement learning (RL) to train refinement *policies* directly from simulation. AMR poses a challenge for RL as both the state dimension and available action set changes at every step, which we solve by proposing new policy architectures with differing generality and inductive bias. The model sizes of these policy architectures are independent of the mesh size and hence can be deployed on larger simulations than those used at training time. We demonstrate in comprehensive experiments on static function estimation and time-dependent equations that RL policies can be trained on problems without using ground truth solutions, are competitive with a widely-used error estimator, and generalize to larger and unseen test problems.

[*]Equal contribution. Correspondence to: Jiachen Yang <yang40@llnl.gov>, Tarik Dzanic <tdzanic@tamu.edu>, Robert Anderson <anderson110@llnl.gov>, Daniel Faissol <faissol1@llnl.gov>

Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS) 2023, Valencia, Spain. PMLR: Volume 206. Copyright 2023 by the author(s).

1 INTRODUCTION

Numerical simulation of PDEs via the finite element method (FEM) (Brenner and Scott, 2007) plays an integral role in computational science and engineering (Reddy and Gartling, 2010; Monk *et al.*, 2003). Given a fixed set of basis functions, the resolution of the finite element mesh determines the trade-off between solution accuracy and computational cost. For problems with large variations in local solution characteristics, uniform meshes can be computationally inefficient due to their suboptimal distribution of mesh density, under-resolving regions with complex features such as discontinuities or large gradients and over-resolving regions with smoothly varying solutions. For systems with multiscale properties, attempting to resolve these features with uniform meshes can be challenging even on the largest supercomputers. To achieve more efficient numerical simulations, adaptive mesh refinement (AMR), a class of methods that dynamically adjust the mesh resolution during a simulation to maintain equidistribution of error, is used to significantly increase accuracy relative to computational cost.

Existing methods for AMR share the iterative process of computing a solution on the current mesh, estimating refinement indicators, marking element(s) to refine, and generating a new mesh by refining marked elements (Bangerth and Rannacher, 2013; Červený *et al.*, 2019). The optimal algorithms for error estimation and marking in many problems, especially evolutionary PDEs, are not known (Bohn and Feischl, 2021), and deriving them is difficult for complex refinement schemes such as hp-refinement (Zienkiewicz *et al.*, 1989). As such, the current state-of-the-art is guided largely by heuristic principles that are derived by intuition and expert knowledge (Zienkiewicz and Zhu, 1992), such as using an instantaneous error estimator with greedy element marking, but choosing the best combination of heuristics is complex and not well understood. Whether and how opti-

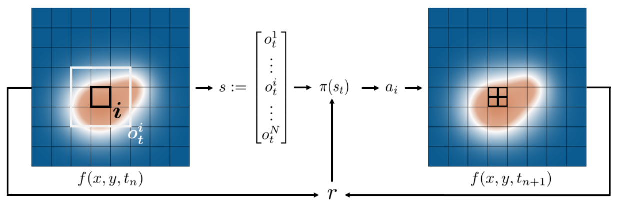


Figure 1: AMR viewed as a Markov decision process.

mal AMR strategies can be found by directly optimizing a long-term performance objective are open questions.

We advance the novel notion that adaptive mesh refinement is fundamentally a *sequential decision-making* problem: making an optimal sequence of refinement decisions to optimize cumulative or terminal accuracy, subject to a computational budget. We hypothesize that a sequence of greedy decisions based on instantaneous error indicators does not constitute an optimal sequence of decisions. This is because the optimality of a current refinement decision depends on complex interactions between solution dynamics, budget consumption, and the propagation of numerical error from the current decision throughout the system, all of which impact solution accuracy many steps into the future and are not revealed by instantaneous error. In time-dependent problems for example, an error estimator by itself cannot preemptively refine elements which would encounter complex features in the next time step.

Given this perspective, we formulate AMR as a Markov decision process (MDP) (Puterman, 2014) (Figure 1) and propose a reinforcement learning (RL) (Sutton and Barto, 2018) approach to train a mesh refinement policy to optimize a performance metric, such as final solution error. AMR poses a challenge for RL as the sizes of the state and set of available actions depend on the current number of mesh elements, which changes with each refinement action at every MDP time step. One may define a fixed bounded state and action space given a finite refinement budget, but this is inefficient as the policy's input-output dimensions must accommodate the full exponentially large space—e.g., input dimensions on the order of millions of degrees of freedom in large applications—whereas only subspaces (with increasing size) are encountered. This motivates us to design efficient policy architectures that leverage the known correspondence between each mesh state and valid actions.

Our paper contributes a proof of feasibility for an entirely novel way of learning AMR strategies using deep RL. In particular: 1) We formally define an MDP with effective variable-size state and action spaces for AMR (Section 3.2); 2) We propose three policy architectures—with differing generality and inductive bias for modeling interaction—that operate on such variable-size spaces (Section 4); 3) Toward the eventual goal of deploying on large and complex problems on which RL cannot tractably be trained, we propose to train on small representative features with known analytic solutions and using a novel reward formulation that applies to problems without known solutions (Section 5); 4) Our experiments demonstrate for the first time that RL can outperform a greedy refinement strategy based on the widely-used Zienkiewicz-Zhu-type error estimator; moreover, we show that an RL refinement policy can generalize to higher refinement budgets and larger meshes, transfer effectively from static to time-dependent problems, and can be effectively trained on more complex problems without readily-available ground truth solutions (Section 6).

2 RELATED WORK

To the best of our knowledge, our work is the first to formulate AMR as a global sequential decision-making problem and show the feasibility of an RL approach. A contemporaneous single-agent local approach centers the decision-making on each individual element (Foucart *et al.*, 2022), requiring different definitions of the environment transition between training and test time for scalability. Brevis *et al.* (2020) apply supervised learning to find an optimal parameterized test space without modifying the degrees of freedom. Bohn and Feischl (2021) show theoretically that the estimation and marking steps of AMR for an elliptic PDE can be represented optimally by a recurrent neural network, but model optimization was not addressed.

Previous work have trained neural networks to predict static (non-adaptive) mesh densities and sizes for use by downstream mesh generators (Dyck *et al.*, 1992; Chedid and Najjar, 1996; Zhang *et al.*, 2020; Pfaff *et al.*, 2020; Chen and Fidkowski, 2020). More recently, neural network policies have been trained via RL to generate a mesh incrementally from initial boundary vertices Pan *et al.* (2023). Recent studies have used graph neural networks (GNN) (Sperduti and Starita, 1997; Gori *et al.*, 2005; Scarselli *et al.*, 2008) to predict PDE dynamics on general unstructured and non-uniform meshes (Alet *et al.*, 2019; Belbute-Peres *et al.*, 2020; Pfaff *et al.*, 2020). Other work use neural networks as function approximators in numerical solvers to achieve faster convergence, generalization, and higher resolution of coarse simulations (Hsieh *et al.*, 2018; Luz *et al.*, 2020; Bar-Sinai *et al.*, 2019). Our work focuses on optimizing a finite element space via training a policy that changes the mesh, rather than predicting dynamics or learning components of a solver.

Adjoint-based methods for goal-oriented AMR can optimize a cost such as terminal solution accuracy (Offermans *et al.*, 2017; Rannacher, 2014; Apel *et al.*, 2014), but they incur significant complexity such as a forward-backward solve and a checkpointing system for the backwards-in-time solution (Becker and Rannacher, 2001), which limits their scalability to coarse grids (Davis and LeVeque, 2020).

Both state and action space sizes change at every time step within an episode in AMR, whereas RL has been typically applied to environments with fixed-size observation and small bounded action spaces in almost all benchmark problems (Mnih *et al.*, 2015; Brockman *et al.*, 2016; Osband *et al.*, 2019). Applications where the available action set varies with state (Berner *et al.*, 2019; Vinyals *et al.*, 2019) do not face the challenge of potentially millions of possible actions that arises in large-scale AMR. Applications of RL to graph-structured problems handle states with increasing size, but the action space does not grow with the size of the graph (You *et al.*, 2018; Trivedi *et al.*, 2020).

3 BACKGROUND AND FORMULATION

3.1 Finite Element Method

Our mesh adaptation strategy is implemented in a FEM-based framework (Brenner and Scott, 2007). In FEM, the domain $\Omega \subset \mathbb{R}^D$ is modeled with a mesh that is a union of E nonoverlapping subsets (*elements*) such that $\Omega := \bigcup \Omega_k$ where $k \in \mathbb{N} : k \leqslant E$ (e.g., see Figure 1). The solution on these elements is represented using polynomials (*basis functions*) which are used to transform the governing equations into a system of algebraic equations via the weak formulation. AMR is a commonly used approach to improve the trade-off between the solution accuracy, which depends on the shape and sizes of elements, and the computational cost, which depends on the number of elements. The most ubiquitous method for AMR is h-refinement, whereby elements are split into smaller elements (refinement) or multiple elements coalesce to form a single element (derefinement).

3.2 AMR as a Markov Decision Process

We formulate AMR with spatial h-refinement[1] as a Markov decision process $\mathcal{M} := (\mathcal{O}, N_{\text{max}}, \mathcal{A}, R, P, \gamma)$ with each component defined as follows. Each episode consists of T RL time steps: for time-dependent PDEs, T spans the entire simulation and there may be multiple underlying PDE evolution steps per RL step; for static problems, T is an arbitrary number of steps at which RL can act. Consider a time step t when the current mesh has $N_t \leq N_{\text{max}} \in \mathbb{N}$ elements. Each element i is associated with an *observation* $o_t^i \in \mathcal{O}$ and the *global state* is $s_t := [o_t^1, \dots, o_t^N] \in \mathcal{O}^{N_t}$. We define $\mathcal{O} := \mathbb{R}^d$ such that each element's observation is a tensor of shape $d := l \times w \times c$ that includes the values and refinement depths of a local window centered on itself (see Appendix A.1.1). For brevity, let $\mathcal{S}_t$ denote the current global state space $\mathcal{O}^{N_t}$. We denote an action by $a_t \in \mathcal{A}_t := \{0, 1, \dots, N_t\} \subset \mathcal{A} := \{0, 1, \dots, N_{\text{max}}\}$, where 0 means "do-nothing" and $i \neq 0$ means refine element i. Given the current state and action, the MDP transition P consists of:

[1]Polynomial p-refinement can be formulated in a similar way. r-refinement (Huang and Russell, 2010; Dobrev *et al.*, 2019) can be formulated as an RL problem but is not treated in this work.

1) refining the selected element into multiple finer elements (which increases N_t) if a refinement budget B is not exceeded and the selected element is not at the maximum refinement depth d_{max};

2) stepping the finite element simulation forward in time (for time-dependent PDEs only);

3) computing a solution on the new finite element space.

Steps 1-3 are standard procedures in FEM (Anderson *et al.*, 2021), knowledge of which is not used in our proposed model-free RL approach. Although the size of the state vector and set of valid actions changes with each time step due to the varying N_t, this MDP is well-defined since one can define the global state space as the union of all $\mathcal{O}^N, N < N_{\text{max}}$, and likewise for the action space. An agent moves through subspaces of increasing size during an episode.

When a true solution is available at *training* time, the reward at step t is defined as the change in error from the previous step, normalized by the initial error to reduce variation across function classes:

$$r_t := (\|e_{t-1}\|_2 - \|e_t\|_2)/\|e_0\|_2 \,, \tag{1}$$

where error e is computed relative to the true solution. With abuse of notation, we shall use e to indicate the error norm. The ground truth is not needed to deploy a trained policy on test problems. When the true solution is not readily available, as is the case for most non-trivial PDEs, one may run a reference simulation on a highly-resolved mesh to compute equation 1, but this approach can be prohibitively expensive for training on large-scale simulations. Instead, we propose the use of a *surrogate reward* $r_t := \|u_{t,\text{refine}} - u_{t,\text{no-refine}}\|_2$, the normed difference between the estimated solution u with and without executing the chosen refinement action. This surrogate, which is an upper bound on the true reward and effectively acts as an estimate of the error reduction, is only used at training time, whereas at test time, the effectiveness of trained policies is evaluated using the error with respect to a highly-resolved reference simulation.

Our objective to find a stochastic policy $\pi \colon \mathcal{S}_t \to \Delta(\mathcal{A}_t)$ to maximize the objective

$$J(\pi) := \mathbb{E}_{a \sim \pi(\cdot|s), s_{t+1} \sim P(\cdot|a, s_t)} \left[\sum_{t=1}^{T} \gamma^t r_t \right] . \tag{2}$$

Aside from $\gamma \in (0, 1)$, the dense and shaped reward (Ng *et al.*, 1999) defined in equation 1 implies that maximizing this objective is equivalent to maximizing total error reduction: $e_0 - e_{\text{final}}$. We work with the class of policy optimization methods as they naturally admit stochastic policies that could benefit AMR at test time: a stochastic refinement action could reveal the need for further refinement in a region that appears flat on a coarse mesh. We build on REINFORCE and PPO (Sutton *et al.*, 2000; Schulman *et al.*, 2017) to train a policy π_θ (parameterized by θ) using batches of trajectories $\{\tau_k := \{(s_t, a_t, r_t)_k\}_{t=1}^T\}_{k=1}^K$ generated by the current policy. By virtue of RL, π_θ is trained not merely to act greedily but to account for dependence of future rewards on current actions.

4 POLICY ARCHITECTURES FOR VARIABLE STATE-ACTION SPACES

The exact 1:1 correspondence between the number of observation components and the number of valid actions calls for dedicated policy architectures for AMR. Different inductive biases expressed by different model architectures can have significant impact on performance even when used in the same learning algorithm (Battaglia *et al.*, 2018). As such, we investigate three policy architectures that address the challenge of variable size state vector $s \in \mathbb{R}^{N_t \times d}$ and action set $\{0, 1, \dots, N_t\}$, where number of elements N_t changes during each episode. These architectures are compatible with any stochastic policy gradient algorithm. We focus on the special case of 1:1 correspondence between the number of observations that compose each global state and the number of available actions at that state. Although not treated in this work, these policy architectures can be easily extended to the general case of 1:k correspondence[2].

4.1 Independent Policy Network

The Independent Policy Network (IPN) handles the 1:1 correspondence by mapping each observation to a probability for the corresponding action. Let $f_\theta \colon \mathbb{R}^d \mapsto \mathbb{R}$ be a function parameterized by θ. Given a matrix of observations $s := [o^1, \dots, o^N] \in \mathbb{R}^{N \times d}$, we define the policy as

$$\pi(\cdot|s) = \texttt{softmax}\left(f_\theta(o^1), \dots, f_\theta(o^N)\right) . \qquad (3)$$

For example, using a neural network with hidden layer $\boldsymbol{W} \in \mathbb{R}^{d \times h}$ with h nodes, output layer $\boldsymbol{H} \in \mathbb{R}^{h \times 1}$, and activation function σ, the discrete probability distribution over N actions conditioned on s is defined by $\texttt{softmax}(\sigma(s\boldsymbol{W})\boldsymbol{H})$. IPN is illustrated in Figure 2a.

IPN applies to meshes of variable size since the set of trainable parameters θ is independent of N, but it has two main limitations. Firstly, it makes a strong assumption of locality as the action probability at an element does not depend on the observations at other elements. This assumption also appears in existing AMR methods that estimate error independently at each element; in fact, the output probabilities of IPN may be viewed as normalized error estimates. Secondly, the permutation equivariance of this architecture—i.e., $\pi(a^{\mu(i)}|(o^{\mu(1)}, \dots, o^{\mu(N)})) = \pi(a^i|s)$ for any permutation operator $\mu \colon [N] \mapsto [N]$—means that one cannot use the ordering of inputs to represent spatial relations among elements, which would be necessary for refining an element based on neighboring conditions. We mitigate this problem by defining each observation as an image that includes neighborhood information and using a convolutional layer, but this may face difficulties on unstructured meshes with non-quadrilateral elements (Červený *et al.*, 2019).

4.2 Hypernetwork Policy

The hypernetwork policy captures higher-order interaction among inputs via the function form (illustrated in Figure 2b)

$$\pi(\cdot|s) = \texttt{softmax}\left(f_{g_\phi(s)}(o^1), \dots, f_{g_\phi(s)}(o^N)\right) . \qquad (4)$$

The main policy network weights θ are now the output of a hypernetwork (Ha *et al.*, 2017) $g_\phi \colon \mathbb{R}^{N \times d} \mapsto \mathbb{R}^{\dim(\theta)}$, parameterized by ϕ, which produces mixing among the inputs $s \in \mathbb{R}^{N \times d}$. Continuing with the example in IPN, where the policy network's first layer is $\boldsymbol{W} \in \mathbb{R}^{d \times h}$, a hypernetwork with two layers can be instantiated as $\left[\sum_{i=1}^N (s\boldsymbol{U})_{i,:}\right] \boldsymbol{V} = \boldsymbol{W}$ where $\boldsymbol{U} \in \mathbb{R}^{d \times h_1}$ and $\boldsymbol{V} \in \mathbb{R}^{h_1 \times (d \times h)}$ are the trainable parameters ϕ, and $\boldsymbol{M}_{i,:}$ denotes the i-th row of matrix $\boldsymbol{M}$. The output $\boldsymbol{W}$ is then used as part of θ in equation 3.

This increased generality comes with more difficulty in the choice of g_ϕ, which affects the extent to which it captures interaction among inputs. It does not contain an inductive bias for the local nature of interactions in physical PDEs. In fact, the use of a summation from $i = 1$ to N in the example above means that complete global information affects each local refinement decision, which is a strong inductive bias.

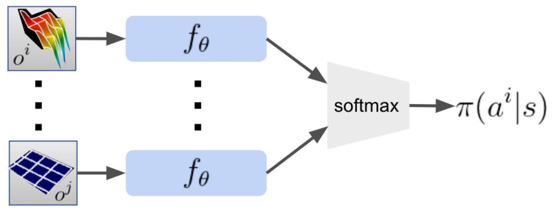
4.3 Graph Network Policy

We build on graph networks (Scarselli *et al.*, 2008; Battaglia *et al.*, 2018) to address both the issue of interaction terms and spatial relation among elements. Specifically, we construct a policy based on Interaction Networks (Battaglia *et al.*, 2016), illustrated in Figure 2c, which is a special case without global attributes[3]. At each step, the mesh is represented as a graph $\mathcal{G} = (V, E)$. Each vertex v^i in $V = \{v^i\}_{i=1:N}$ corresponds to element i and is initialized to be the observation o^i. $E = \{(e^k, r^k, s^k)\}_{k=1:N^e}$ is a set of edges with attributes e^k between sender vertex s^k and receiver vertex r^k. An edge exists between two vertices if and only if they are spatially adjacent. We define the initial edge attribute e^k as a one-hot vector indicator of the difference in refinement depth between r^k and s^k.

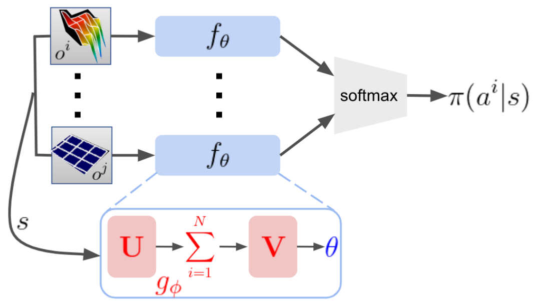
Graph networks capture the relations between nodes and edges via the inductive bias of its update rules. The model we use consists of a learned edge network ϕ^e and a node

[2]To include de-refinement actions ($k = 2$), the only change is to map each element's observation to two output logits, one interpreted as refinement and the other de-refinement, and include both in the global `softmax` over all elements.

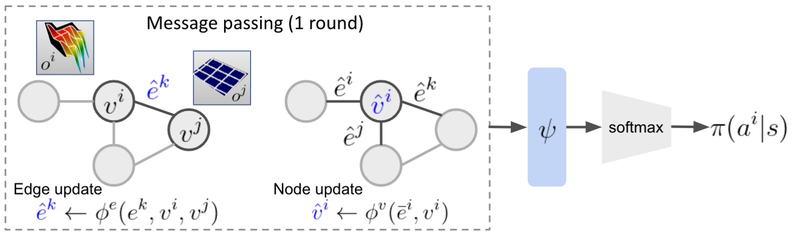
[3]While not demonstrated in this work, one may define a global graph attribute containing the PDE coefficients or initial/boundary conditions and apply variants of Graph Networks that use global attributes in their update rules to improve generalization.



(a) Independent Policy Network



(b) Hypernetwork policy



(c) Graph network policy

Figure 2: RL policy architectures for adaptive mesh refinement. (a) Independent Policy Network (IPN) applies the same function f_θ to map each element's observation o^i independently to logits, which are then mapped to action probabilities via a global `softmax`. (b) The Hypernetwork policy generates the parameters θ of the main network via a hypernetwork g_ϕ that receives global state s. (c) The Graph Network policy conducts multiple rounds of message passing, using learned edge network ϕ^e and node network ϕ^v to update edge and node features individually and in parallel for all edges and nodes. Final node features are mapped to logits, on which a global `softmax` is applied.

network ϕ^v, which are applied individually and in parallel to all edges and nodes. A single *forward pass* through the graph policy involves multiple rounds of *message passing* (see Algorithm 1). Each round is defined by the following operations: 1) Each edge attribute e^k is updated by learned function φ^e using local node information via $\hat{e}^k \leftarrow \varphi^e(e^k, v^{r^k}, v^{s^k})$; 2) For each node i, we denote by $\hat{E}^i := \{(\hat{e}^k, r^k, s^k)\}_{r^k=i}$ the set of all edges with node i as the receiver, and all updated edge attributes are aggregated into a single feature $\bar{e}^i \leftarrow \rho^{e\to v}(\hat{E}^i)$ by aggregation function $\rho^{e\to v}$ (e.g., element-wise sum); 3) Then, each node attribute is updated by $\hat{v}^i \leftarrow \varphi^v(\bar{e}^i, v^i)$ using learned function φ^v. Each round increases the size of the neighborhood that determines node attributes. Finally, we map each node attribute to a scalar using learned function ψ, apply a global softmax over all nodes, and interpret the value at each node i as the probability of choosing element i for refinement.

The graphnet policy addresses both limitations of the IPN and the hypernetwork policy. Cross terms arise in the forward pass due to mutual updates of edge and node attributes using local information. The order of cross terms increases with each message-passing round. Local spatial relations between mesh elements are included by construction in the adjacency matrix and initial edge attributes, so there is no need to include global spatial information in each element's observation vector.

5 EXPERIMENTAL SETUP

Our experiments assess the ability of RL, using the proposed policy architectures, to find AMR strategies that generalize to test function[4] classes that differ from the training class, generalize to variable mesh sizes and refinement budgets, and extend to more complex problems without readily-available ground truth solutions. We define the FEM environment in Section 5.1, the train-test procedure in Section 5.2, and the implementation of our method and baselines in Section 5.3.

5.1 AMR Environment

MFEM. We use MFEM (Anderson *et al.*, 2021; MFEM, 2020), a modular open-source C++ library for FEM, to implement the MDP for AMR. We ran experiments on two classes of AMR problems: static and time-dependent. In the static case, the objective of mesh refinement is to minimize the L^2 error norm of projecting a variety of test functions onto a two-dimensional H^1 finite element space. In the time-dependent case, the functions are projected onto a two-dimensional L^2 finite element space, and a PDE of the form $\frac{\partial u}{\partial t} + \nabla\cdot\mathbf{F}(u) = 0$ is solved on a periodic domain using the finite element framework. Unlike the static case, the numer-

[4]In the sense of testing a policy's performance after training, not in the sense of the weak formulation of PDEs.

ical error accumulated at each time step propagates with the physical dynamics and determines future error. Two types of PDEs were used: the linear advection equation, where $\mathbf{F}(u) = \mathbf{c}u$ with $\mathbf{c} = [1, 0]$, and the nonlinear Burgers equation, where $\mathbf{F}(u) = \mathbf{c}u^2$ with $\mathbf{c} = [1, 0.3]$. The advection equation is used as there is an analytic solution to provide a ground truth, whereas the Burgers equation is used as a representative of more complex physical systems, including shock and rarefaction waves, without a readily-available analytic solution. The solution is represented using continuous (or discontinuous) second-order Bernstein polynomials for the static (or time-dependent) case, and the initial mesh is partitioned into $n_x \times n_y$ quadrilateral elements. Appendix A.1.1 contains further on the FEM and MDP implementations.

True solutions. We defined a collection of parameterized function classes, each exhibiting features such as sharp discontinuities and smooth variations, from which we randomly sample ground truth functions $f\colon [0,1]^2 \mapsto \mathbb{R}$ to initialize each episode. The collection, shown in Figure 3 and defined precisely in Appendix A.1.2, includes: *bumps*, *circles*, *steps*, and *steps2* (a combination of two steps). These functions with closed form allow us to compute the error and reward at train time for static and advection problems. In the case of Burgers equation where the exact solution is not readily-available, we either use reference simulations on a highly-resolved mesh to act as a "ground truth" or employ the surrogate reward (defined in Section 3.2) to compute the reward for training.

In the static case, the true solution is fixed and each simulation time step is an RL step. For the time-dependent PDE cases, the initial solution is transported through the periodic domain and the ratio of simulation time steps to RL steps is set such that a feature advecting at unit velocity returns to its original position after 10 RL steps. We set episode length at training time to equal the refinement budget B, with $B = 10$ for static problems, $B = 20$ for advection, and $B = 50$ for Burgers. Larger budget (i.e., longer episode) was chosen for the time-dependent problems to test the ability of RL to find long-term refinement strategies that outperform greedy baselines. All methods were subject to the same budget constraint. Due to the Gibbs phenomena in FEM, using smooth polynomial approximations to solve hyperbolic systems containing discontinuities can introduce spurious oscillations which, in turn, can cause the simulation to become unstable. Therefore, we limit the true solutions to smooth functions (e.g., *bumps*, *circles*) for the advection and Burgers cases. Due to the nonlinearity of Burgers equation, initially smooth solutions can develop discontinuities in finite time. This behavior is resolved using the flux-corrected transport (FCT) approach (Boris and Book, 1997).

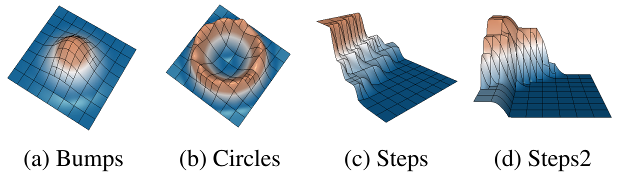


Figure 3: Examples from each true solution function class.

5.2 Experiments and Performance Metric

We conducted the following experiments to compare RL policies with baselines:

In-distribution: Train and test on true solutions sampled from the same function class. Training and testing on different function classes is unlikely in applications, since expert knowledge of solution features is almost always available and one can train on similar functions, but we include experimental results in Appendix B.

Out-of-distribution: For problems such as Burgers equation where training on multiple initial conditions (ICs) may be expensive, we show the effectiveness of policies trained on a single initial condition (IC) when tested on multiple random ICs, either with or without fine-tuning.

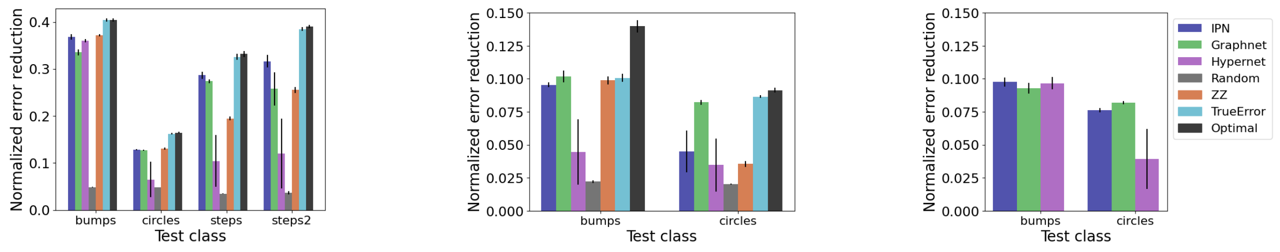
Generalization: 1) **Static→advection**: Policies trained on static functions are tested on advection. 2) **Budget↑**: Policies trained with a small refinement budget B (20 on static and 10 on advection) are test with $B = 50, 100$. 3) **Size↑**: Policies trained on an 8×8 mesh are tested on meshes up to size 200×200 (360k solution nodes), a 625x increase in element number, with and without preserving the relative solution and mesh length scales.

We define the performance of a given refinement policy in an episode in the static case as $(e_{\text{initial}} - e_{\text{final}})/e_{\text{initial}}$, where e_{initial} (or e_{final}) is the error norm at the beginning (or end) of an episode, to remove the variation in the error due to different true solution classes and random function initialization within each class. In the time-dependent case, without any refinement, the error may increase over the course of the simulation due to the accumulation of discretization error. Hence, we define performance as $(e_{\text{no-refine, final}} - e_{\text{final}})/e_{\text{initial}}$, where $e_{\text{no-refine, final}}$ is the final error without any refinement.

For every experiment and every policy architecture, we trained four independent policies with different random seeds. For each test case, we report the mean and standard error—over the four independent policies and with different simulator seeds —of the mean performance metric over 100 test episodes. At each test episode, all methods faced the same initial condition (which differs across episodes).

5.3 Implementation and Baselines

We describe the high-level implementation here and provide complete details in Appendix A.2. All policy architectures



(a) Static (b) Advection (c) Static→advection

Figure 4: **In-distribution** and **Static→advection**. Performance of IPN, Graphnet and Hypernetwork policies versus baselines. Higher values are better. (a,b) RL policies were trained and tested on the same function class, for static and advection cases independently. (c) Static-trained policies on a function class are tested on advection of the same class.

use a convolutional neural network with the same architecture as the input layer. The **IPN** has two fully-connected hidden layers with h_1 and h_2 nodes and `ReLU` activation, followed by a `softmax` output layer. Its action on input states is described in Section 4.1. The **Graphnet** policy is implemented with the Graph Nets library (Battaglia *et al.*, 2018). Each input state consists of node observation tensors, all edge vectors, and the adjacency matrix. Node tensors are first passed through an Independent block, after which multiple Interaction networks (Battaglia *et al.*, 2016) act on both node and edge embeddings to produce a probability at each node (see Section 4.3). The **Hypernet policy** is parameterized by matrices $\boldsymbol{U} \in \mathbb{R}^{d \times h_1}$, $\boldsymbol{V} \in \mathbb{R}^{h_1 \times (d \times h)}$, and $\boldsymbol{Y} \in \mathbb{R}^{d \times h}$, where h_1 and h are design choices. $\boldsymbol{U}$ and $\boldsymbol{V}$ act on input state s to produce the main policy weights $\boldsymbol{W} \in \mathbb{R}^{d \times h}$ while $\boldsymbol{Y}$ acts on s to produce a bias $b \in \mathbb{R}^h$, so that the main policy's first hidden layer is $\texttt{ReLU}(s\boldsymbol{W} + b)$. Output probabilities are computed in the same way as IPN.

Baselines. The **ZZ** policy uses a Zienkiewicz-Zhu-type recovery-based error estimator (Zienkiewicz and Zhu, 1992) and refines the element with the largest estimated error. The **TrueError** policy refines the element where the error of the numerical solution with respect to the true solution is largest. It is effectively an upper bound on the performance of any state-of-the-art method based on instantaneous error estimator, but it is *not* the theoretical upper bound on performance because refining the element with largest current error does not necessarily result in smallest final error. It cannot be deployed without known solutions. The **GreedyOptimal** policy performs one-step lookahead by checking all possible outcomes of refining each element individually and chooses the element whose refinement would result in the lowest error. It is intractable even for simple time-dependent PDEs on relatively coarse meshes. TrueError and GreedyOptimal are strong *oracles* that cannot be used in real applications.

6 RESULTS

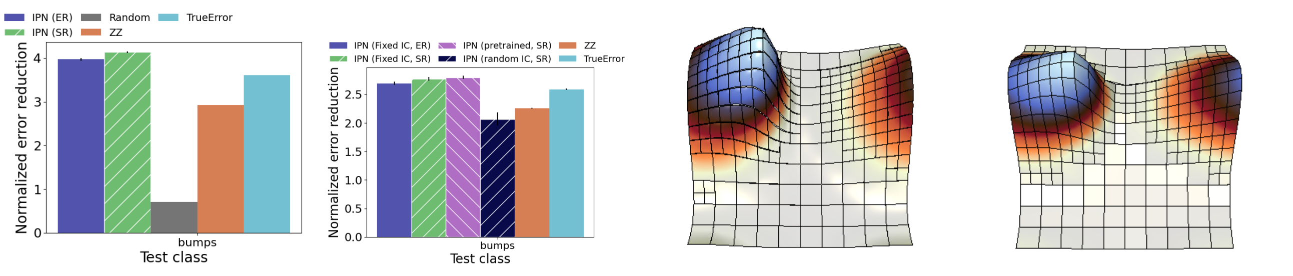
We find that the proposed methods achieve performance that is competitive with baselines, outperforming or matching ZZ on 20 out of 24 cases while being competitive with (even sometimes outperforming) the *oracle* TrueError strategy. More importantly, RL policies generalize well to larger refinement budgets and mesh sizes, and transfer effectively from a static problem to a time-dependent problem. Almost always, all methods produced the same number of mesh elements at each simulation time step, which means that differences in performance are solely due to differences in refinement strategies. Appendix A.5 provides information on the cost of training and decision times. Videos of policies on advection and Burgers can be viewed at `https://sites.google.com/view/rl-for-amr`.

6.1 In-distribution

Static functions (Figure 4a). RL policies either meet or significantly exceed the performance of ZZ on all function classes. Notably, both IPN and Graphnet outperform ZZ significantly on *steps* by spending the limited refinement budget only on regions with discontinuities (Figure 3c). On the smoother function classes such as *bumps* where ZZ is known to perform well, all three policy architectures have comparable performance to ZZ. Overall, IPN outperforms both Graphnet and Hypernet. This suggests that capturing higher-order interaction among observations, each of which already contains local neighborhood information, is unnecessary for estimation of static functions as they only have a local domain of influence. Hypernetwork policies converged to the behavior of making no refinements on at least one out of four independent runs on all classes except *bumps*. This could be attributed to the inherent difficulty of choosing and training a highly nonlinear model.

Advection (Figure 4b). As explained above, we limit the true solutions to smooth functions (*bumps* and *circles*) in the advection case. Graphnet significantly outperformed ZZ on *circles* and is comparable to TrueError on *bumps*, while IPN is comparable to ZZ on both functions. Hypernet is comparable to ZZ on *circles* but has high variance across independent runs. Graphnet's higher performance than other methods indicates that its inductive bias can better represent the local geometric relations between neighboring mesh elements along the circle.

Burgers equation. In experiments with a single bump func-



(a) Test on fixed IC (b) Test on random ICs (c) Mesh by IPN (Fixed IC, SR) (d) Mesh by IPN (pretrained, SR)

Figure 5: **Burgers equation** and **surrogate reward**. Solid bars/ER denote exact reward, striped bars/SR denote surrogate reward. (a) IPN trained and *tested on a fixed IC*. (b) IPN *tested on random ICs* using policies trained on a fixed IC (from Figure 5a), policies pretrained on fixed IC and fine-tuned on random ICs, and policies only trained on random ICs. (c/d) Visualization of resulting meshes for Burgers equation with a fixed bump IC at $T = 50$.

tion (visualized in Figure 5c) as the fixed initial condition (IC) in both train and test, IPN trained with both the exact and surrogate rewards outperformed all baselines (Figure 5a). Surprisingly, the policy trained using the surrogate reward slightly outperformed the policy trained with the exact reward, indicating that: 1) the surrogate reward can be effectively used to train policies without the need for a ground truth solution, as is necessary for general random ICs; 2) because the surrogate reward provides a positive reward whenever a refinement action causes a change in the solution, it effectively acts as an “exploration bonus”, which has been observed in the RL literature to improve performance (Tang *et al.*, 2017). Significantly, Figure 5d and the linked videos show the learned policy generates a moving refinement “front” that anticipates and tracks a moving region that requires refinement.

6.2 Out-of-distribution (OOD)

Figure 5b shows the performance of IPN RL policies and baselines on Burgers equation with random ICs. Policies trained on a fixed IC using either the exact or surrogate reward (“Fixed IC, ER” and “Fixed IC, SR”) generalize well to random unseen ICs and still outperform baselines. Moreover, policies that were pretrained with the surrogate reward on the fixed IC for 2k episodes and fine-tuned with the surrogate reward on random ICs for another 2k episodes performed the best (“pretrained, SR”). Policies trained only on random ICs with the surrogate reward were not as performant, indicating that the training time was not sufficient and that pretraining on a fixed IC is a more efficient approach.

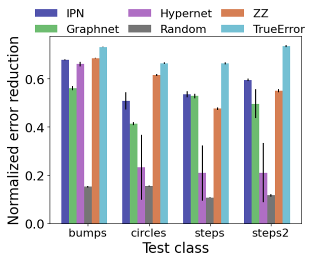
6.3 Generalization

The proposed architectures enable a trained policy to generalize well on a test mesh of different sizes and with different budget constraints, because they map from local element features to element selection probabilities. As long as local features on the test mesh continue to resemble those seen in training, the trained policy continues to perform even when the global test function was previously unseen.

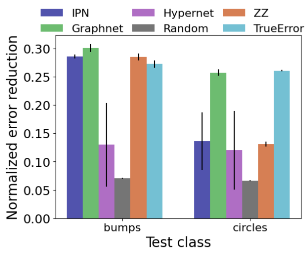
Static→advection (Figure 4c). All static-trained policies demonstrated comparable performance to ZZ and TrueError when tested on *advection-bumps*, while both IPN and Graphnet significantly outperformed ZZ on *advection-circles*. Surprisingly, static-trained IPN significantly outperforms advection-trained IPN when tested on *advection-circles*, and the static-trained Hypernet does so as well on *advection-bumps*, while static-trained Graphnet maintains comparable performance to its advection-trained counterpart (Figure 4b vs. Figure 4c). Figure 14 shows that a static-trained policy on *bumps* with $B = 10$ correctly refines the region of propagation on *advection-bumps* with $B = 50$.

Budget↑ (Figure 6). RL policies trained with low refinement budget generalize to test cases with higher budget. In the static case, comparing Figure 4a ($B = 10$) with Figure 6a ($B = 50$) shows that the performance of RL policies relative to ZZ is generally preserved by the increase in refinement budget. Figures 6 and 9 show that an IPN trained with $B = 10$ makes qualitatively correct refinement decisions when allowed $B = 100$ during test. In the advection case (Figure 6b), Graphnet trained with $B = 20$ significantly outperforms both ZZ and TrueError when tested with $B = 50$ on *bumps* and comes within the margin of error of TrueError on *circles*. Figure 8 shows that an IPN trained with $B = 20$ correctly allocates a higher budget $B = 100$ to the limited region of propagation.

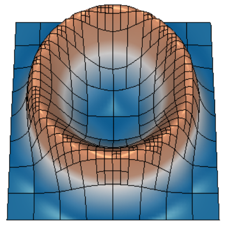
Size↑ (Figure 7). In the static case, the relative performance of RL policies that were trained with an 8×8 mesh (Figure 4a) is generally preserved when deployed on a 16×16 mesh (Figure 7a). All policy architectures outperform ZZ on *bumps*, while IPN and Graphnet still outperform ZZ on *steps*. IPN and Graphnet were comparable to ZZ on 8×8 but underperformed on 16×16 on *circles*. Nonetheless, Figure 11 shows that IPN makes qualitatively correct refinements. On advection, relative performance is preserved on *circles* while IPN and Graphnet deproved slightly on *bumps* (Figure 4b vs. Figure 7b). Without preserving the solution-to-mesh length scales, the prior tests emulate deploying a policy trained on



(a) Static B=50



(b) Advection B=50


(c) IPN on static circle

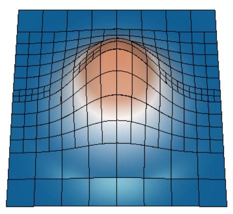
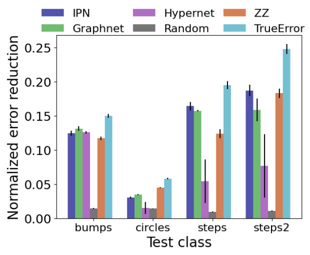
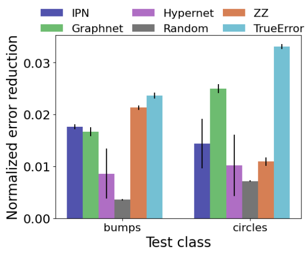

(d) Graphnet on advection

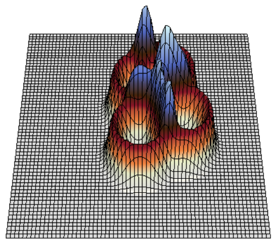
Figure 6: **Budget**↑. (a-b) Policies trained with budget B=10 (static) and B=20 (advection) are tested with B=50. (c) IPN trained with B=10 generalizes to B=100. (d) Graphnet trained on advecting bump with B=20 generalizes to B=50.

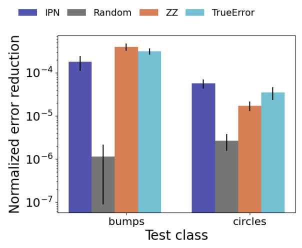


(a) Static 16x16



(b) Advection 16x16


(c) Circles on 64x64 mesh



(d) Advection 200x200

Figure 7: **Size**↑. (a-b) Policies trained on 8×8 mesh were tested on 16×16 mesh. (c-d) Policies trained on 8×8 mesh were tested on (c) 64×64 and (d) 200×200 meshes with approximately constant feature-to-mesh length scale ratio.

coarser versions of the simulations. When tested on 64×64 and 200×200 meshes (64 and 625-fold increase in number of mesh elements, respectively, e.g. Figure 7c) that preserve the solution-to-mesh length scales, Figures 7d and 10 show that IPN is competitive with baselines on bumps and even outperforms TrueError on circles. This emulates training a policy on a small subset of a highly-resolved simulation and deploying it on the full simulation.

6.4 Choice of Architecture

For a given test problem, one can choose among the three proposed architectures based on the type of local features that are anticipated to arise. Such prior knowledge is available for many practical classes of problems that have a long history of instances that were solved by traditional methods. Comparing overall performance across architectures, Figures 4a, 6a and 7a show that the IPN is the best candidate for problems whose local features are stationary or slowly-propagating (as in the Burgers experiments), whereas Figures 4b, 6b and 7b show that the Graphnet policy is the best candidate for advection problems with smooth features.

7 LIMITATIONS

Our current approach is limited to refinement of one element per MDP time step. For practical applications, one can define a “refinement step” to consist of multiple MDP time steps, so that multiple elements are refined before the solution is updated and the PDE advances in simulation time. De-refinement was not included as it requires a multi-objective optimization treatment (error versus cost) that detracts from the main purpose of this work. Although our results show the RL can outperform greedy baselines, we only conjecture that this is due to anticipatory refinement and leave a deeper investigation to future work. The scale and complexity of experiments in this work were chosen for agile demonstration of feasibility and generalization, and we hope these promising results serve as a milestone for future application to more complex problems.

8 CONCLUSION

We contributed a proof of feasibility for scalable application of reinforcement learning for adaptive mesh refinement. Our experiments on static and time-dependent problems demonstrate that RL policies can outperform the widely-used ZZ-type error estimator, outperform an oracle based on true error, generalize to different refinement budgets and larger meshes, transfer from static to time-dependent settings, and generalize to more complex problems even when trained without ground truth rewards. Our finding that RL policies sometimes outperform the true error baseline supports the hypothesis that instantaneous error-based strategies are not optimal due to their inability to refine preemptively. Future work can extend our methods to include derefinement actions, take a multi-agent perspective, and tackle the unification of $h-$, $p-$, and $r-$ refinement.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 and the LLNL-LDRD Program with project tracking #21-SI-001. LLNL-CONF-820002.

References

Alet, F., Jeewajee, A. K., Villalonga, M. B., Rodriguez, A., Lozano-Perez, T., and Kaelbling, L. (2019). Graph element networks: adaptive, structured computation and memory. In *International Conference on Machine Learning*, pages 212–222. PMLR.

Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Cerveny, J., Dobrev, V., Dudouit, Y., Fisher, A., Kolev, T., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., and Zampini, S. (2021). MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*, **81**, 42 – 74. Development and Application of Open-source Software for Problems with Numerical PDEs.

Apel, T., Pfefferer, J., and Rösch, A. (2014). Graded meshes in optimal control for elliptic partial differential equations: an overview. *Trends in PDE constrained optimization*, pages 285–302.

Bangerth, W. and Rannacher, R. (2013). *Adaptive finite element methods for differential equations*. Birkhäuser.

Bar-Sinai, Y., Hoyer, S., Hickey, J., and Brenner, M. P. (2019). Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, **116**(31), 15344–15349.

Battaglia, P., Pascanu, R., Lai, M., Rezende, D. J., *et al.* (2016). Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, pages 4502–4510.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., *et al.* (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Becker, R. and Rannacher, R. (2001). An optimal control approach to a posteriori error estimation in finite element methods. *Acta numerica*, **10**(1), 1–102.

Belbute-Peres, F. d. A., Economon, T., and Kolter, Z. (2020). Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In *International Conference on Machine Learning*, pages 2402–2411. PMLR.

Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., *et al.* (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

Bohn, J. and Feischl, M. (2021). Recurrent neural networks as optimal mesh refinement strategies. *Computers & Mathematics with Applications*, **97**, 61–76.

Boris, J. P. and Book, D. L. (1997). Flux-corrected transport. *Journal of Computational Physics*, **135**(2), 172–186.

Brenner, S. and Scott, R. (2007). *The mathematical theory of finite element methods*, volume 15. Springer Science & Business Media.

Brevis, I., Muga, I., and van der Zee, K. G. (2020). Data-driven finite elements methods: Machine learning acceleration of goal-oriented computations. *arXiv preprint arXiv:2003.04485*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. *arXiv preprint arXiv:1606.01540*.

Červený, J., Dobrev, V., and Kolev, T. (2019). Nonconforming mesh refinement for high-order finite elements. *SIAM Journal on Scientific Computing*, **41**(4), C367–C392.

Chedid, R. and Najjar, N. (1996). Automatic finite-element mesh generation using artificial neural networks-Part I: Prediction of mesh density. *IEEE Transactions on Magnetics*, **32**(5), 5173–5178.

Chen, G. and Fidkowski, K. (2020). Output-based error estimation and mesh adaptation using convolutional neural networks: Application to a scalar advection-diffusion problem. In *AIAA Scitech 2020 Forum*, page 1143.

Davis, B. N. and LeVeque, R. J. (2020). Analysis and performance evaluation of adjoint-guided adaptive mesh refinement for linear hyperbolic pdes using clawpack. *ACM Transactions on Mathematical Software (TOMS)*, **46**(3), 1–28.

Dobrev, V., Knupp, P., Kolev, T., Mittal, K., and Tomov, V. (2019). The Target-Matrix Optimization Paradigm for high-order meshes. *SIAM Journal on Scientific Computing*, **41**(1), B50–B68.

Dyck, D., Lowther, D., and McFee, S. (1992). Determining an approximate finite element mesh density using neural network techniques. *IEEE transactions on magnetics*, **28**(2), 1767–1770.

Foucart, C., Charous, A., and Lermusiaux, P. F. (2022). Deep reinforcement learning for adaptive mesh refinement. *arXiv preprint arXiv:2209.12351*.

Gori, M., Monfardini, G., and Scarselli, F. (2005). A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734. IEEE.

Ha, D., Dai, A., and Le, Q. V. (2017). Hypernetworks. In *International Conference on Learning Representations*.

Hsieh, J.-T., Zhao, S., Eismann, S., Mirabella, L., and Ermon, S. (2018). Learning neural pde solvers with convergence guarantees. In *International Conference on Learning Representations*.

Huang, W. and Russell, R. D. (2010). *Adaptive moving mesh methods*, volume 174. Springer Science & Business Media.

Luz, I., Galun, M., Maron, H., Basri, R., and Yavneh, I. (2020). Learning algebraic multigrid using graph neural networks. In *International Conference on Machine Learning*, pages 6489–6499. PMLR.

MFEM (2020). MFEM: Modular finite element methods [Software]. `https://mfem.org`.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., *et al.* (2015). Human-level control through deep reinforcement learning. *Nature*, **518**(7540), 529.

Monk, P. *et al.* (2003). *Finite element methods for Maxwell's equations*. Oxford University Press.

Ng, A. Y., Harada, D., and Russell, S. J. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann Publishers Inc.

Offermans, N., Peplinski, A., Marin, O., and Schlatter, P. (2017). Adjoint error estimators and adaptive mesh refinement in nek5000.

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepesvari, C., Singh, S., *et al.* (2019). Behaviour suite for reinforcement learning. In *International Conference on Learning Representations*.

Pan, J., Huang, J., Cheng, G., and Zeng, Y. (2023). Reinforcement learning for automatic quadrilateral mesh generation: A soft actor–critic approach. *Neural Networks*, **157**, 288–304.

Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., and Battaglia, P. (2020). Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*.

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Rannacher, R. (2014). Model reduction by adaptive discretization in optimal control. *Trends in PDE Constrained Optimization*, pages 251–284.

Reddy, J. N. and Gartling, D. K. (2010). *The finite element method in heat transfer and fluid dynamics*. CRC press.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2008). The graph neural network model. *IEEE Transactions on Neural Networks*, **20**(1), 61–80.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sperduti, A. and Starita, A. (1997). Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, **8**(3), 714–735.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, X., Duan, Y., Schulman, J., De Turck, F., and Abbeel, P. (2017). # exploration: A study of count-based exploration for deep reinforcement learning. In *31st Conference on Neural Information Processing Systems (NIPS)*, volume 30, pages 1–18.

Trivedi, R., Yang, J., and Zha, H. (2020). Graphopt: Learning optimization models of graph formation. In *International Conference on Machine Learning*, pages 9603–9613. PMLR.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., *et al.* (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, **575**(7782), 350–354.

You, J., Liu, B., Ying, Z., Pande, V. S., and Leskovec, J. (2018). Graph convolutional policy network for goal-directed molecular graph generation. In *NeurIPS*.

Zhang, Z., Wang, Y., Jimack, P. K., and Wang, H. (2020). Meshingnet: a new mesh generation method based on deep learning. In *International Conference on Computational Science*, pages 186–198. Springer.

Zienkiewicz, O., Zhu, J., and Gong, N. (1989). Effective and practical h–p-version adaptive analysis procedures for the finite element method. *International Journal for Numerical Methods in Engineering*, **28**(4), 879–891.

Zienkiewicz, O. C. and Zhu, J. Z. (1992). The superconvergent patch recovery and a posteriori error estimates. Part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, **33**(7), 1331–1364.

Algorithm 1 Graphnet policy forward pass

```
1:  for each message-passing round do
2:      for $k \in \{1, \dots, N^e\}$ do
3:          $\hat{e}^k \leftarrow \varphi^e(e^k, v^{r^k}, v^{s^k})$ # Update edge attribute
4:      end for
5:      for $i \in \{1, \dots, N\}$ do
6:          $\hat{E}^i := \{(\hat{e}^k, r^k, s^k)\}_{r^k=i}$ # Edge set for $v^i$
7:          $\bar{e}^i \leftarrow \rho^{e \to v}(\hat{E}^i)$ # Aggregation for $v^i$
8:          $\hat{v}^i \leftarrow \varphi^v(\bar{e}^i, v^i)$ # Update vertex attribute
9:      end for
10:     $e^k \leftarrow \hat{e}^k, \forall k \in [N^e]$, $v^i \leftarrow \hat{v}^i, \forall i \in [N]$
11: end for
12: $\mathbb{R} \ni x^i \leftarrow \psi(v^i), \forall i \in [N]$
13: $\pi(a^i|s)$ is the $i$-th entry of softmax$(x^1, \dots, x^N)$
```

A Experimental setup

A.1 Environment details

A.1.1 MFEMCtrl

To interface between the MFEM framework and the RL environment, we developed MFEMCtrl, a C++/Python wrapper for the AMR and FEM capabilities in MFEM. MFEMCtrl is used to convert solutions to observations, apply refinement decisions, and calculate errors.

The initial mesh is partitioned into $n_x \times n_y = 8 \times 8$ elements for static and advection experiments and 10×10 for Burgers equation. Generalization experiments on larger initial mesh used $n_x \times n_y = 16 \times 16$ or 64×64. The true solution is projected onto the finite element space by interpolation to the nodes of the Bernstein basis functions. After each refinement action, the solution is projected again onto the refined mesh (for the static case) or integrated in time until the next refinement action (for the time-dependent case). The maximum refinement depth is fixed by the parameter d_{max} such that the maximally-refined mesh consists of $2^{d_{\text{max}}} n_x \times 2^{d_{\text{max}}} n_y$ elements. d_{max} was set to 3 for static experiments, whereas $d_{\text{max}} = 2$ for advection and $d_{\text{max}} = 1$ for Burgers equation due to the time step restrictions imposed by the Courant-Friedrichs-Lewy (CFL) condition of the finest elements.

Observation. The observation consisted of the solution and the depth of each element. Since the gradients of the solution are, by definition, a function of the solution, the observation does not include the gradients as they can be implicitly learned. The solution/depth of each element was observed by interpolating the functions to a local equispaced mesh (*image*) centered around each element, shown by the white box in Figure 1. Each element's observation is a $l \times w \times c$ tensor where $l = w = l_{\text{element}} + 2l_{\text{context}}$ is the spatial observation window with $l_{\text{element}} = 16$ sampled points inside the element and $l_{\text{context}} = 4$ sampled points in a coordinate direction outside the element. We chose $c = 2$ channels so that estimated function values and element depths are observed, while gradients are omitted since the policy network can in principle estimate gradients from the value channel. To impose a 1:1 map between each observation and possible action, we append a dummy o^0 to state s corresponding to action 0. At most one refinement is allowed per MDP step.

Table 1: Parameterized true solutions

	Parameter	[min, max]
Bumps (static)	c_x	[0.2, 0.9]
	c_y	[0.2, 0.9]
	w	[0.05, 0.2]
	n	$\{1, \ldots, 6\}$
Bumps (advection)	c_x	[0.3, 0.7]
	c_y	[0.3, 0.7]
	w	[0.005, 0.05]
	n	$\{1, \ldots, 4\}$
Bumps (Burgers) single IC	c_x	0.5
	c_y	0.5
	w	0.05
	n	1
Bumps (Burgers) random IC	c_x	[0.3, 0.7]
	c_y	[0.3, 0.7]
	w	[0.005, 0.05]
	n	$\{1, \ldots, 4\}$
Circles (static)	c_x	[0.2, 0.8]
	c_y	[0.2, 0.8]
	r	[0.05, 0.2]
	w	[0.1, 1.0]
	n	$\{1, \ldots, 6\}$
Circles (advection)	c_x	[0.3, 0.7]
	c_y	[0.3, 0.7]
	r	[0.05, 0.2]
	w	[0.03, 0.05]
	n	$\{1, \ldots, 4\}$
Steps and Steps2	o	[0, 1.0]
	θ	$[0, \pi/2]$
	n	$\{1, \ldots, 6\}$

A.1.2 Ground truth functions

Bumps

$$\begin{aligned}
n &\sim \text{Uniform}[n_{\text{min}}, n_{\text{max}}] \\
c_{x,i} &\sim \text{Uniform}[c_{x,\text{min}}, c_{x,\text{max}}], \quad i = 1, \ldots, n \\
c_{y,i} &\sim \text{Uniform}[c_{y,\text{min}}, c_{y,\text{max}}], \quad i = 1, \ldots, n \\
w_i &\sim \text{Uniform}[w_{\text{min}}, w_{\text{max}}], \quad i = 1, \ldots, n \\
f(x, y) &= \sum_{i=1}^{n} \exp\left(-\frac{(x - c_{x,i})^2 + (y - c_{y,i})^2}{w_i}\right)
\end{aligned}$$

Circles

$$\begin{aligned}
n &\sim \text{Uniform}[n_{\text{min}}, n_{\text{max}}] \\
c_{x,i}, c_{y,i} &\sim \text{Uniform}[c_{\text{min}}, c_{\text{max}}], \quad i = 1, \ldots, n \\
r_i &\sim \text{Uniform}[r_{\text{min}}, r_{\text{max}}], \quad i = 1, \ldots, n \\
w_i &\sim \text{Uniform}[w_{\text{min}}, w_{\text{max}}], \quad i = 1, \ldots, n \\
f(x, y) &= \sum_{i=1}^{n} \exp(\\
&\quad - \frac{(\sqrt{(x - c_{x,i})^2 + (y - c_{y,i})^2} - r_i)^2}{w_i})
\end{aligned}$$

Steps

$$\begin{aligned}
n &\sim \text{Uniform}[n_{\text{min}}, n_{\text{max}}] \\
\theta &\sim \text{Uniform}[\theta_{\text{min}}, \theta_{\text{max}}] \\
o_i &\sim \text{Uniform}[o_{\text{min}}, o_{\text{max}}], \quad i = 1, \ldots, n \\
f(x, y) &= \sum_{i=1}^{n} 1 + \tanh\left[100(o_i - (x + y \tan\theta))\right]
\end{aligned}$$

Steps2

$$\begin{aligned}
n &\sim \text{Uniform}[n_{\text{min}}, n_{\text{max}}] \\
\theta_i &\sim \text{Uniform}[\theta_{\text{min}}, \theta_{\text{max}}], \quad i = 1, \ldots, n \\
o_i &\sim \text{Uniform}[o_{\text{min}}, o_{\text{max}}], \quad i = 1, \ldots, n \\
s_i &:= (x - 0.5)\cos\theta_i - (y - 0.5)\cos\theta_i \\
f(x, y) &= \frac{1}{2} \sum_{i=1}^{n} 1 + \tanh\left[100(s_i - o_i)\right]
\end{aligned}$$

A.1.3 Reward

Let u_t denote the true solution at time t, let $\hat{u}_t$ denote the estimated solution on the mesh at time t. For a given mesh at time $t-1$, a given time evolution of the true solution from $t-1$ to t, and a refinement action a_t (which may be "do-nothing"), let $\hat{u}_{t,\text{refine}}$ denote the estimated solution on the mesh at time t that has undergone refinement action a_t, and let $\hat{u}_{t,\text{no-refine}}$ denote the estimated solution on the mesh at time t without that refinement. We have two reward definitions:

1. Delta norm reward with true solution

$$r_t := (e_{t-1} - e_t)/e_0 \tag{5}$$
$$e_t := \|u_t - \hat{u}_t\|_2 \tag{6}$$

2. Surrogate reward

$$r_t := \|\hat{u}_{t,\text{refine}} - \hat{u}_{t,\text{no-refine}}\|_2 \tag{7}$$

We used the first reward definition for static and advection experiments where analytic true solutions are available, and for Burgers experiments involving a single initial condition and pre-computed reference data that acts as the true solution. We used the second reward definition for all other Burgers experiments.

A.2 Implementation

We used standard policy gradient (Sutton *et al.*, 2000) for all experiments except for experiments on Burgers equation and generalization of 8x8-trained advection policies to 64x64 test meshes. We used PPO (Schulman *et al.*, 2017) for the latter two cases. We trained for 20k episodes on static problems, 10k episodes on advection problems, 2k episodes on Burgers equation with a single IC, and 4k episodes on Burgers equation with random ICs. The Burgers experiment with pretraining used 2k episodes on a single IC and a further 2k on random ICs. Each episode is initialized with refinement budget B, where $B = 10$ for static problems, $B = 20$ for advection, and $B = 50$ for Burgers.

IPN. For efficient computation on a batch of B trajectories, where each trajectory b consists of T environment steps and each step t_b consists of a variable-sized global state $s \in \mathbb{R}^{N_{t_b} \times d}$, we merge the variable dimension with the batch and time dimension to form an input matrix whose dimensions are $[\sum_{b=1}^{B} \sum_{t=1}^{T} N_{t_b}, d]$. The output is reshaped into a "ragged" matrix of logits with dimensions $[B \times T, N_{t_b}]$, where the row lengths vary for each batch and time step. A softmax operation over each row produces the final action probabilities at each step.

Graphnet policy. The first graph layer is an Independent recurrent block that passes the input node tensors through a convolutional layer followed by a fully-connected layer, to arrive at node embeddings. This is followed by two recurrent passes through an InterationNetwork (Battaglia *et al.*, 2016) where fully-connected layers are used for edge and node update functions. A final InteractionNetwork output layer followed by a global softmax over the graph produces a scalar at each node, which is interpreted as the probability of selecting the corresponding element for refinement. Except for the input node feature $v^i \in \mathbb{R}^d$ and output node scalar, all internal node (edge) embeddings have the same size, denoted as dim(v) (dim(e)). We fixed dim$(e) = 16$ for both static and advection and tuned dim(v) (Table 2).

Hypernet policy. We fixed the main network's hidden layer dimension at $h = 64$ and tuned the hypernetwork's hidden layer dimension h_1 (Table 2).

A.3 Hyperparameters

For both static and advection problems, we tuned a subset of all hyperparameters for all methods by the following procedure to handle the large set of policy architectures and ground truth functions. Chosen values of tuned hyperparameters are given in Table 2; all other hyperparameters have the same values for all methods and are listed below. We conducted tuning in a multi-task setup, where we train a single policy on functions randomly sampled from all ground truth classes, with randomly sampled parameters according to Appendix A.1.2. This is done separately on static and advection problems. The tuning process is coordinate descent where the best parameter from one sweep is used for the next sweep. We start with exploration decay $\epsilon_{\text{div}} \in \{100, 500, 1000, 5000\}$ (a lower bound on exploration was enforced by using behavioral policy $\tilde{\pi}(a_t|s_t) = (1-\epsilon)\pi(a_t|s_t) + \epsilon/N_t$ with ϵ decaying linearly from ϵ_{start} to ϵ_{end} by ϵ_{div} episodes). Next we tune the size of hidden layers in the policy network (over $(h_1, h_2) \in \{(128, 64), (256, 64), (128, 128), (256, 256)\}$ for IPN, node representation dimension $\dim(v) \in \{32, 64, 128, 256\}$ for Graphnet, and $h_1 \in \{16, 32, 64, 128\}$ for Hypernet). Lastly, we tune the learning rate $\alpha \in \{5 \cdot 10^{-5}, 10^{-4}, 5 \cdot 10^{-4}, 10^{-3}, 5 \cdot 10^{-3}\}$. For Graphnet and Hypernet, we inherit the best ϵ_{div} from IPN because optimal exploration depends in large part on the complexity of the environment, which is the same across all policy architectures.

Separately for the static and advection cases, all three policy architectures have the same values for all other hyperparameters. These are: policy gradient batch size 8, initial exploration lower bound $\epsilon_{\text{start}} = 0.5$, final exploration lower bound $\epsilon_{\text{end}} = 0.05$, discount factor $\gamma = 0.99$, convolutional neural network layer with 6 filters of size $(5, 5)$ and stride $(2, 2)$.

Table 2: Hyperparameters for IPN, Graphnet and Hypernet policies on static and advection AMR.

	Static			Advection		
Parameter	IPN	Graphnet	Hypernet	IPN	Graphnet	Hypernet
ϵ_{div}	500	500	500	100	100	100
IPN (h_1, h_2)	(128, 64)	-	-	(256,256)	-	-
Graphnet $\dim(v)$	-	64	-	-	256	-
Hypernet h_1	-	-	128	-	-	32
α	10^{-4}	10^{-4}	$5 \cdot 10^{-5}$	10^{-4}	10^{-4}	10^{-4}

For Burgers experiments and advection experiments on generalization from 8×8 to 64×64 initial mesh sizes, we used a more comprehensive population-based hyperparameter search with successive elimination for all methods. We start with a batch of n_{batch} tuples, where each tuple is a combination of hyperparameter values, with each value sampled either log-uniformly from a continuous range or uniformly from a discrete set. We train independently with each tuple for n_{episode} episodes, eliminate the lower half of the batch based on their final performance, then initialize the next set of n_{episode} episodes with the current models for the remaining tuples. We use the hyperparameters of the last surviving model. Chosen values are shown in Table 3.

The hyperparameter ranges are: discount factor in $\{0.1, 0.5, 0.99\}$, policy entropy coefficient in $(10^{-3}, 1.0)$, GAE λ in $\{0.85, 0.90, 0.95\}$, learning rate in $(10^{-5}, 5 \cdot 10^{-3}$, PPO ϵ in $(0.01, 0.5)$, value loss coefficient in $\{0.1, 0.5, 1.0\}$, IPN h_1 in $\{128, 256\}$, and IPN h_2 in $\{64, 128, 256\}$.

Table 3: Hyperparameters for advection size $\uparrow$ and Burgers experiments

	Advection (IPN)		Burgers (IPN)	
Parameter	Bumps	Circles	1 IC	Random IC
Discount γ	0.99	0.99	0.1	0.5
Entropy coefficient	0.0133	0.0689	$8.84 \cdot 10^{-3}$	$1.17 \cdot 10^{-3}$
GAE λ	0.95	0.9	0.85	0.85
Learning rate	$1.18 \cdot 10^{-3}$	$4.8 \cdot 10^{-3}$	$1.59 \cdot 10^{-3}$	$2.11 \cdot 10^{-4}$
PPO ϵ	0.0113	0.195	0.128	0.169
Value loss coefficient	0.1	0.5	0.5	0.1
IPN h_1	128	256	256	128
IPN h_2	128	128	128	256

A.4 Scale of experiments

The scale and scope of our current experimental setup is comparable with previous work at the intersection of FEM/PDE and machine learning, whose experimental settings are the following:

1. 2D Poisson equation with a 64x64 square train mesh Hsieh *et al.* (2018)
2. diffusion PDE on 2D triangular mesh with number of nodes ranging from 1024 to 400k (Luz *et al.*, 2020)
3. models of simulations with 1k-5k nodes (Pfaff *et al.*, 2020)

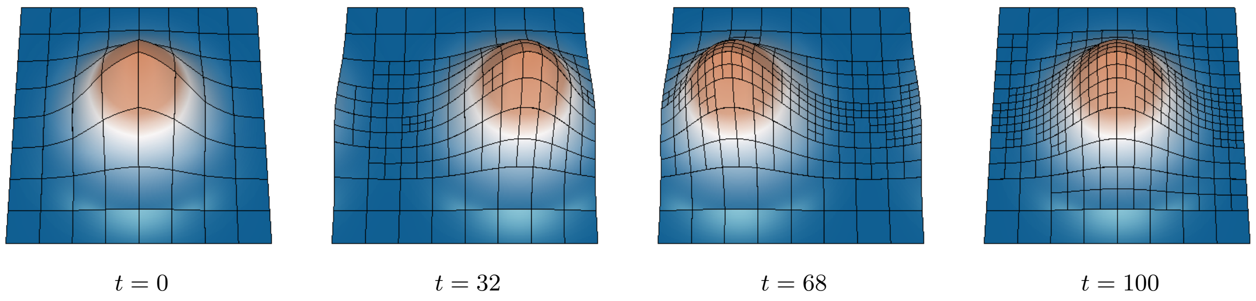


Figure 8: Advection of a bump function. RL policy trained with budget $B = 20$ generalizes to $B = 100$.

4. Poisson equation on 7x7 mesh with square and sphere domains (Alet *et al.*, 2019)
5. airfoil with 6648 nodes on a fine mesh and 354 nodes on a coarse mesh (Belbute-Peres *et al.*, 2020)

A.5 Computing infrastructure and runtime

Experiments were run on Intel 8-core Xeon E5-2670 CPUs, using one core for each independent policy training session. Average training time with 20k episodes in the static case was approximately 6 hours for IPN and Hypernet, and 9 hours for Graphnet. Average training time with 10k episodes in the advection case was approximately 14 hours for IPN and Hypernet, and 18 hours for Graphnet. In general, the one-shot training time for RL is justified by considering that: 1) given a high-performing trained policy, the time incurred in training is negligible when amortized over all future deployment of that policy; 2) the end-to-end RL approach of optimizing policies directly with experience in simulation may provide a faster path toward new refinement policies that can integrate h-, p-, and r-refinement into a unified strategy, whereas developing new AMR techniques is very human-labor intensive, so the training time of RL policies would be a negligible expense.

Table 4 shows that test runtimes for all practically-deployable methods are comparable and within the same order of magnitude. We take the mean over (10 episodes) * (10 steps per episode), which accounts for the impact of variability of different element refinements on the solver time. The total differences in runtime between the RL approach and the various baselines manifest only through the differences in the runtime of evaluating the refinement indicator—the RL policy or the baselines—which is reported in Table 4. This is because the time required for computing a solution in between refinement actions is nearly identical between all approaches, as the meshes for all methods are of identical size at each step and there are negligible differences in the computational cost required to solve the PDE on two meshes that differ by only the choice of which elements are refined. Degrees of freedom (DoF) cost of RL and baselines were the same since all methods refine one element per MDP step in our main experiments.

Table 4: Mean (standard error) time in milliseconds per refinement decision on various initial mesh partitions.

	8×8	16×16	24×24	100×100
IPN	3.22 (0.07)	5.85 (0.05)	9.64 (0.28)	158 (5)
Graphnet	7.74 (0.33)	13.9 (0.43)	23.7 (0.19)	1445 (6)
Hypernet	8.08 (0.08)	10.7 (0.05)	14.3 (0.17)	151 (9)
ZZ	1.96 (0.01)	6.94 (0.01)	15.5 (0.05)	134 (3)

B Additional results

For time-dependent problems at training time, only one element is refined for each step that the PDE advances in time. However, at test time, we can execute the policy multiple times to refine multiple elements before the PDE advances in time. Figure 13 shows that the policy makes appropriate choices of 20 elements per step.

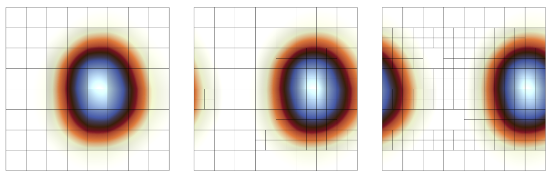
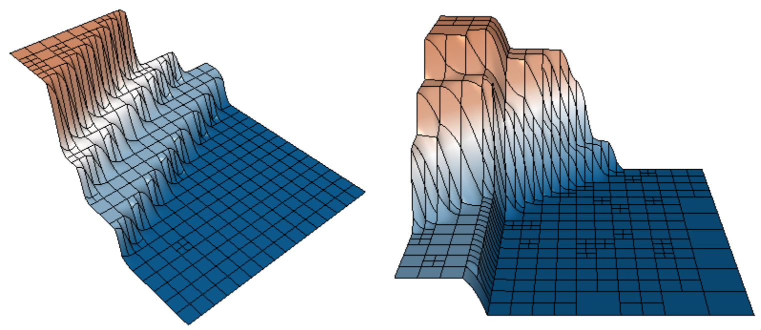


Figure 13: Multiple refinements per solver step.



(a) Steps

(b) Steps2

Figure 9: Generalization of policies trained with refinement budget $B = 10$ to test case with $B = 100$.

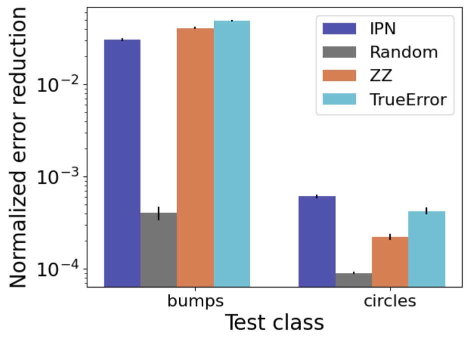
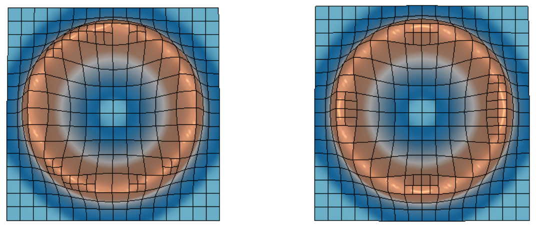


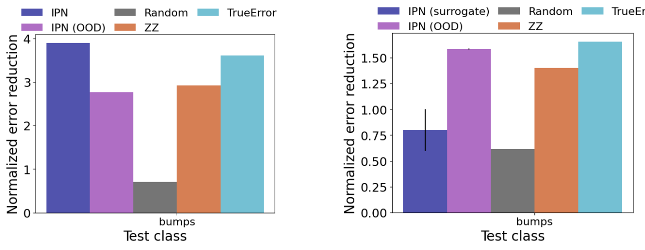
Figure 10: Generalization of policy trained on 8×8 mesh to 64×64 mesh.



(a) IPN

(b) ZZ

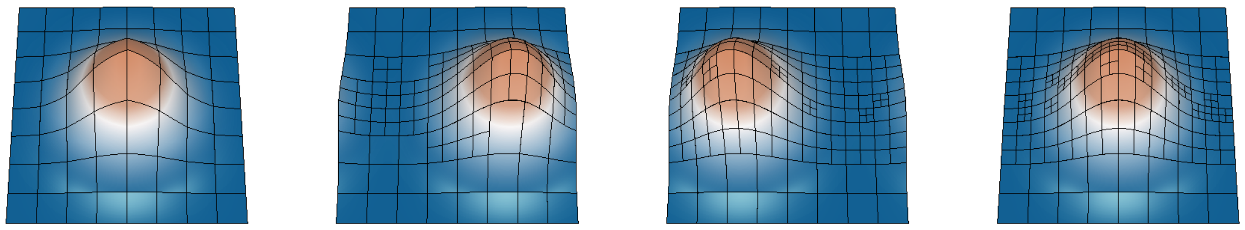
Figure 11: IPN trained on 8×8 initial mesh underperformed ZZ when tested on 16×16 initial mesh but makes qualitatively correct refinements.



(a) 1-bump Burgers

(b) Multi-bumps Burgers

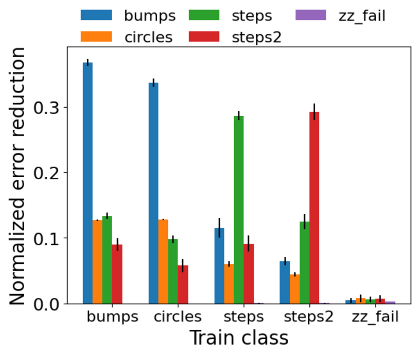
Figure 12: **OOD**. (a) IPN (OOD) was trained on Burgers with multi-bumps IC and tested on Burgers with a 1-bump IC. (b) IPN (OOD) was trained on 1-bump IC and tested with multi-bumps IC.



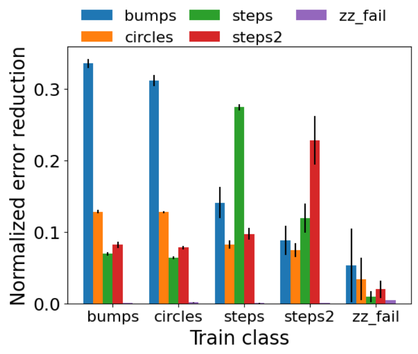
$t = 0$ $t = 22$ $t = 38$ $t = 50$

Figure 14: **Static→advection** and **Budget↑**: IPN trained on static bumps ($B = 10$) transfers to advection ($B = 50$).

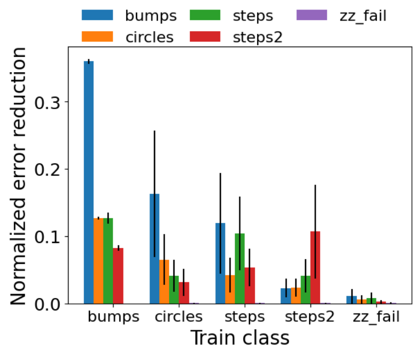
OOD. In the static case (Figures 15a to 15c), IPN policies trained on *circles* transfer well to *bumps* (and vice versa). Hypernet policies performed poorly overall even in the case of **in-distribution**, and consequently does not show comparable performance when transferring across function classes. In the advection case (Figures 15d to 15f), both IPN and Graphnet policies trained on *bumps* significantly outperformed ZZ when tested on *circles* (compare to ZZ in Figure 4b).



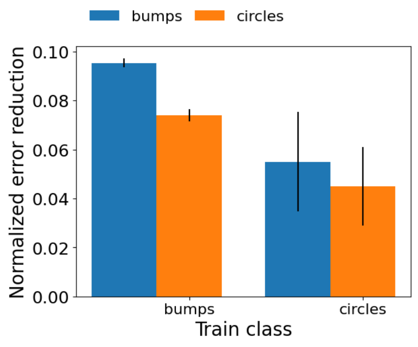
(a) Static IPN



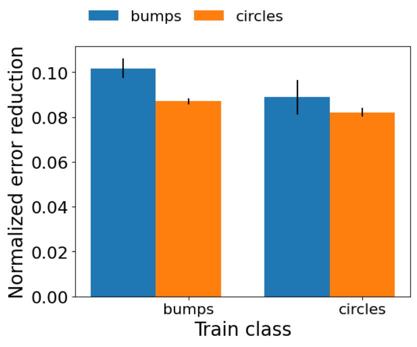
(b) Static Graphnet



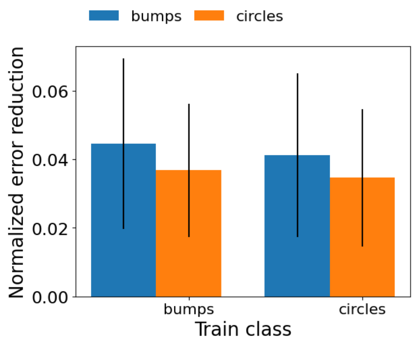
(c) Static Hypernet



(d) Advection IPN



(e) Advection Graphnet



(f) Advection Hypernet

Figure 15: All train-test combinations. Normalized error reduction of IPN, Graphnet and Hypernetwork policies on (a-c) Static AMR and (d-f) Advection PDE. Higher values are better. Legend (colors) shows test classes. RL policies were trained and tested on each combination of true solutions. Mean and standard error over four RNG seeds of mean final error over 100 test episodes per method.