Reinforcement Learning for Adaptive Mesh Refinement

Jiachen Yang [*1] **Tarik Dzanic** [*2] **Brenden Petersen** [*3] **Jun Kudo** [*3] **Ketan Mittal** [3] **Vladimir Tomov** [3]
Jean-Sylvain Camier [3] **Tuo Zhao** [1] **Hongyuan Zha** [1] **Tzanio Kolev** [3] **Robert Anderson** [3] **Daniel Faissol** [3]

Abstract

Large-scale finite element simulations of complex physical systems governed by partial differential equations 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 new problem for RL in that 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 scale to arbitrarily large and complex simulations. We demonstrate in comprehensive experiments on static function estimation and the advection of different fields that RL policies can be competitive with a widely-used error estimator and generalize to larger, more complex, and unseen test problems.

1. Introduction

Numerical simulation of partial differential equations (PDEs) via the finite element method (FEM) (Brenner & Scott, 2007) plays an integral role in computational science and engineering (Reddy & Gartling, 2010; Zienkiewicz & Taylor, 2005; 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 complex systems with large variations in local solution characteristics, uniform meshes can be

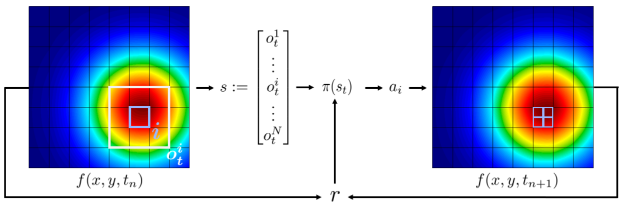


Figure 1: AMR viewed as a Markov decision process.

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 multi-scale properties in particular, such as cosmological hydrodynamics (Snaith et al., 2018), geophysical flows (Burstedde, 2013), and plasma physics (Fujimoto, 2018), 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 same iterative process of computing a solution on the current mesh, estimating refinement indicators, marking element(s) to refine, and generating a new mesh by refinement of marked elements (Bangerth & Rannacher, 2013; Červený et al., 2019). The optimal algorithms for error estimation and marking in many problems, especially time-dependent PDEs, are not known (Bohn & Feischl, 2019), and deriving them is difficult especially 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, but choosing the best combination of heuristics is complex and not well understood. The scalable heuristic of pairing an effective error estimator, such as the widely-used Zienkiewicz-Zhu error estimator (Zienkiewicz & Zhu, 1992), with greedy element selection at each step does not directly aim for long-term optimality over the entire simulation. On the other hand, goal-oriented methods (Becker & Rannacher, 2001) require complex adjoint solvers and checkpointing mechanisms that limit their generality.

[*]Equal contribution [1]Georgia Institute of Technology [2]Texas A&M University [3]Lawrence Livermore National Laboratory. Correspondence to: Daniel Faissol <faissol1@llnl.gov>.

Preprint. Under review.

We posit that adaptive mesh refinement is fundamentally a sequential decision-making problem in which a sequence of greedy decisions based on instantaneous error indicators does not constitute an optimal sequence of decisions for the actual goal of achieving high cumulative or terminal accuracy. For example, in time-dependent problems such as advection, an error estimator by itself cannot preemptively refine elements which would encounter complex features in the next time step. Moreover, the numerical error accumulated at the current time step will itself propagate throughout the physical system and determine how the error at future time steps will accumulate. This means that the optimality of a refinement decision depends on the accuracy of future states and that selecting an element which yields the largest reduction in error at the current time step may not be the optimal decision over the entire simulation. Whether and how optimal AMR strategies can be found by directly optimizing a long-term performance objective are open questions.

Given this viewpoint, we formulate AMR as a Markov decision process (MDP) (Puterman, 2014) (Figure 1) and propose a reinforcement learning (RL) (Sutton & Barto, 2018) approach that explicitly trains a mesh refinement policy to optimize a performance metric, such as minimizing final solution error. In contrast to most, if not all, benchmark problems and complex applications of RL (Mnih et al., 2015; Brockman et al., 2016; Osband et al., 2019; Berner et al., 2019; Vinyals et al., 2019), AMR poses a new challenge for learning a refinement policy: the sizes of both the input state and the set of available actions depend on the current number of mesh elements, which changes with each refinement action at every MDP time step. While one may define a fixed and bounded state and action space given a finite refinement budget, doing so is very inefficient as the policy model's input-output dimensions would have to accommodate the full exponentially large space but only subspaces (with increasing size) are encountered during simulation. In many practical applications, one would routinely encounter input dimensions on the order of millions or billions of degrees of freedom. This motivates the design of efficient policy architectures that leverage the known correspondence between the current mesh state and valid action set.

We make the following conceptual, methodological, and experimental contributions:

⋆ 1) we formally define an MDP with variable-size state and action spaces for AMR (Section 3.2);

⋆ 2) we propose three policy architectures—with differing generality, inductive bias, and capacity for modeling interaction—that operate on such state-action spaces of arbitrary and variable size (Section 4);

⋆ 3) as a path toward solving large and complex problems on which RL cannot tractably be trained, we propose the experimental approach of training on small representative features with known analytic solutions and investigating the generalizability of resulting policies;

⋆ 4) our experiments on static function estimation and advection problems demonstrate for the first time that RL can be competitive with, and sometimes outperform, a greedy refinement strategy based on the widely-used Zienkiewicz-Zhu error estimator; moreover, we show that an RL refinement policy can generalize to higher refinement budgets and initial mesh densities, and transfer effectively from static to time-dependent problems (Section 6).

2. Related work

The formulation of problems in numerical analysis as statistical learning problems can be traced at least as far in time as to Poincaré (Poincaré, 1912; Diaconis, 1988). Contemporary works have employed neural networks as powerful function approximators in existing numerical PDE and linear system solvers to achieve faster convergence rates, generalize to different boundary conditions or larger problems, and approximate underresolved features in coarse-grained simulations (Hsieh et al., 2019; Luz et al., 2020; Bar-Sinai et al., 2019). Our work focuses on optimizing a finite element space rather than components of a numerical solver.

To the best of our knowledge, no prior work has formulated adaptive mesh refinement as a sequential decision-making problem and proposed a reinforcement learning approach (Sutton & Barto, 2018). Previous work at the intersection of neural networks and mesh-based simulation trained neural networks to predict mesh densities, sizes, or error fields for use by downstream mesh generators (Dyck et al., 1992; Chedid & Najjar, 1996; Zhang et al., 2020; Pfaff et al., 2020; Chen & Fidkowski, 2020). Brevis et al. (2020) apply supervised learning to find an optimal parameterized test space without modifying the degrees of freedom. Bohn & Feischl (2019) 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 left as an open question.

Recent studies have leveraged the effectiveness of graph neural networks (GNN) (Sperduti & Starita, 1997; Gori et al., 2005; Scarselli et al., 2008) at representing relational structure to predict PDE dynamics on general unstructured and non-uniform meshes (Alet et al., 2019; Belbute-Peres et al., 2020; Pfaff et al., 2020). Previous work on graph generation and formation have employed GNNs as the policy model in an RL context with applications to biological and social network datasets (You et al., 2018; Trivedi et al., 2020). To our knowledge, no prior work has investigated the effectiveness of GNN-based policies for AMR.

Learning a policy for unbounded variable-size state and action spaces is a new problem for RL, which 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). While there are notable applications where the available action set varies with state (Berner et al., 2019; Vinyals et al., 2019), they do not face the challenge of potentially millions of possible actions that arises in large-scale AMR. The curriculum learning technique of growing action spaces (Farquhar et al., 2020) maintains a fixed action space size within each episode. It is orthogonal to the AMR problem where both state and action space sizes change at every time step within an episode.

3. Background and notation

3.1. Finite element method

Our mesh adaptation strategy is implemented in a FEM-based framework (Brenner & 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 $k \in \mathbb{N} : k \leqslant E$. 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.

For a given problem, it is well known that the accuracy of the solution depends on the shape and size of the elements, and the computational cost of the solution process depends on the number of elements in the mesh. AMR is a commonly used approach to maximize the accuracy of the solution and/or minimize the computational cost of the calculation. The most ubiquitous method for AMR is h-refinement, in which the shape of elements is modified by splitting an element into smaller elements (refinement) or coalescing multiple elements to form a single element (derefinement). In practical applications where an exact solution is not known, these refinement decisions are typically made using *a posteriori* error estimators which rely on the numerical solution and its derived quantities obtained on a given mesh. The conventional AMR approach with *a posteriori* error estimators is to refine the element(s) with the highest estimated error based on the *current* solution. Therefore, these decisions are based on a greedy approach without regard to the optimality of future refinement actions.

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. 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 $s_t := [o_t^1, \ldots, o_t^N] \in \mathcal{O}^{N_t}$. 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, refinement depths and (optionally) the gradients of a local window centered on itself. For brevity, let $\mathcal{S}_t$ denote the current global state space $\mathcal{O}^{N_t}$. We denote an action by $a_t \in \mathcal{A}_t := \{0, 1, \ldots, N_t\} \subset \mathcal{A} := \{0, 1, \ldots, N_{\text{max}}\}$, where we let 0 denote the do-nothing action[2]. Given the current state and action, the MDP transition P consists of:

⋆ 1) refining the selected element into multiple fine 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.

The reward at step t is defined as the change in error compared to the previous step, normalized by the initial error to reduce variation across classes of training functions:

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

where error e is computed relative to a ground truth solution. The ground truth is only used in training and not needed to deploy a trained policy on new test problems. With abuse of notation, we shall use e to indicate the error norm. Our objective to find a stochastic policy $\pi \colon \mathcal{S}_t \to \Delta(\mathcal{A}_t)$ to maximize the objective

$$J(\pi) := \mathbb{E}_{a \sim \pi(\cdot|s), s_{t+1} \sim P(\cdot|a,s_t)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] . \tag{2}$$

Ignoring the discount factor $\gamma \in (0,1)$, this objective is equivalent to maximizing total error reduction: $e_0 - e_{\text{final}}$.

Despite the fact that the size of the state vector and set of valid actions changes with each time step due to the varying number of elements N_t, the MDP is well-defined since one can define the global state space as the set of all possible $\mathcal{O}^N$, $N < N_{\text{max}}$, and likewise for the action space. Nonetheless, to our knowledge, such variation in state-action spaces does not occur in any existing RL application. Moreover, the exact 1:1 correspondence between the number of observation components and the number of valid actions calls for designing a dedicated policy architecture for AMR, which we present below in Section 4.

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 (e.g., Figure 2c). We build on the policy gradient algorithm (Sutton et al., 2000) to train a policy π_θ (parameterized by θ) with Monte Carlo returns using batches of trajectories $\{\tau_b := \{(s_t, a_t, r_t)_k\}_{t=1}^{T}\}_{k=1}^{K}$ generated by the current policy.

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

[2] To impose a 1:1 map between each observation and possible action, we append a dummy o^0 to s corresponding to action 0. At most one refinement is allowed per MDP step.

4. Policy architectures for variable state-action spaces

We propose three policy architectures that directly address the challenge of variable size state vector $s \in \mathbb{R}^{N_t \times d}$ and action set $\{0, 1, \dots, N_t\}$, both of whose sizes changes with number of elements N_t within an episode. These architectures are compatible with any stochastic policy gradient algorithm. Our exposition focuses 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, it is straightforward to adapt these policy architectures to the general case of 1:k correspondence—e.g., $k = 2$ whereby each element may be refined or derefined.

We make two remarks for conceptual clarity. Firstly, h-refinement constructs a refinement tree in simulation but we do not employ this structure in our methods as there is no indication from traditional AMR algorithms, which do not store solutions at coarse elements, that hierarchical information is useful for optimal refinement (Červený et al., 2019). Secondly, the existence of a refinement tree has no *a priori* relation to the class of Monte Carlo Tree Search (MCTS) algorithms, which can optionally be combined with, but is orthogonal to, RL (Silver et al., 2017).

4.1. Independent policy network

The first policy architecture, called 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

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

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

This policy architecture scales to meshes of any 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 i does not involve any "cross terms" that depend on the observations at elements $j, k \neq i$. 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 its behavior is independent of the ordering of input element observations. This prevents one from using the ordering of inputs to represent spatial relations among elements, which would be necessary for a policy to refine an element based on neighboring conditions. We address this problem by defining each element's observation as an image tensor that includes neighborhood information and using a convolutional network layer, but this may face difficulties on meshes with non-quadrilateral elements and general manifold meshes.

4.2. Hypernetwork policy

We build on hypernetworks (Ha et al., 2017) to generalize IPN to include higher-order interaction among inputs via the function form

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

The main policy network weights θ are now the output of a hypernetwork $g_\phi \colon \mathbb{R}^{N \times d} \mapsto \mathbb{R}^{\dim(\theta)}$, 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 $W \in \mathbb{R}^{d \times h}$, a hypernetwork with two layers can be instantiated as

$$\left[\sum_{i=1}^{N} (sU)\,[i,:]\right] V = W \, , \qquad (5)$$

where $U \in \mathbb{R}^{d \times h_1}$ and $V \in \mathbb{R}^{h_1 \times (d \times h)}$ are the trainable parameters ϕ, and $M[i,:]$ denotes the i-th row of matrix M. The output W can then be used as part of θ in (3).

The hypernetwork policy's increased generality comes at a cost: the extent to which it captures interaction among inputs depends on its architecture, which is difficult to choose in advance. It does not contain an inductive bias for the local nature of interactions seen in applications of AMR to simulations of classical physics. In fact, the use of a summation from $i = 1$ to N over the input dimension in the example above means that complete global information affects each local refinement decision, which may not be a good inductive bias. It also inherits the difficulty of IPN in capturing spatial information.

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), which is a special case without global attributes[3]. At each step, the mesh is represented as a graph $G = (V, E)$. 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

[3]Global attributes arise in supervised learning applications such as classification of a dataset of graphs. While not used in this work, it is possible to use function coefficients and initial/boundary conditions as global attributes in a graphnet policy for AMR.

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 internal update rules. A single *forward pass* through the graph policy involves one or many rounds of *message passing* (Algorithm 1 in the Appendix). Each round is defined by the following set of simultaneous computations:

$\star$ 1) each edge attribute e^k is updated by learned function φ^e using local node information via $\hat{e}^k \leftarrow \varphi^e(e^k, v^{r^k}, v^{s_k})$;

$\star$ 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 $\bar{e}^i \leftarrow \rho^{e\rightarrow v}(\hat{E}^i)$ by aggregation function $\rho^{e\rightarrow v}$ (e.g., element-wise sum);

$\star$ 3) then the node attribute is updated by $\hat{v}^i \leftarrow \varphi^v(\bar{e}^i, v^i)$ using learned function φ^v. Each additional round increases the size of the local neighborhood that determines node attributes. Finally, we map each node attribute to a scalar using learned function ψ and apply a global softmax over all nodes and interpret the value at each node i as the probability of choosing element i for refinement.

These update rules allow the graphnet policy to solve 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 initial edge attributes, so there is no need to include numerical spatial information in each element's observation vector.

5. Experimental setup

Our experiments assess the ability of RL with the proposed policy architectures to find AMR strategies in comparison to canonical error estimators, generalize to test function classes that differ from the train class, and generalize to variable mesh sizes and refinement budgets. 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. Results are analyzed in Section 6.

5.1. AMR environment

MFEM. We use MFEM (Anderson et al., 2021; MFEM), a modular open-source C++ library for FEM, to implement the MDP for AMR. The proposed methods are evaluated on two classes of AMR problems: static and time-dependent. In the static case, a variety of test functions (true solutions) are projected onto a two-dimensional finite element space, and a sequence of mesh refinements is performed to minimize the L^2 norm of the projection error onto the domain $[0,1]^2$. Training RL policies to solve static problems is not

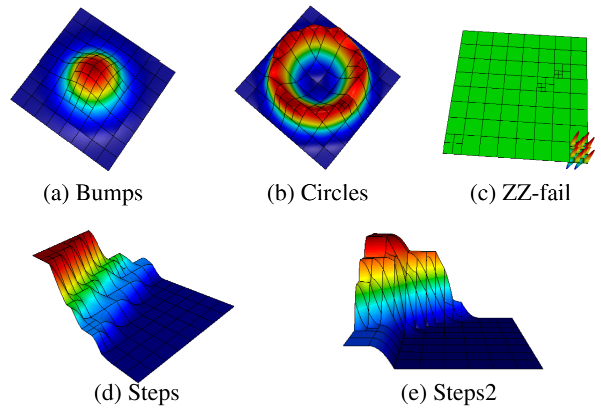


Figure 2: Individual samples from each true solution function class. Each function sampled in *bumps* and *circles* is a superposition of a random number of such features in general. Refinements shown here are produced by IPN.

an end in itself; instead, we posit and experimentally verify that policies trained on static features can be effectively deployed on time-dependent problems whose dynamics produce such features. In the time-dependent case, the functions are projected onto the finite element space, and the linear advection equation, given by

$$\frac{\partial u}{\partial t} + \vec{c} \cdot \vec{\nabla} u = 0,$$

where $\vec{c} = [1, 0]$ denotes the velocity vector, is solved on a periodic domain using the finite element framework. The advection equation is used as there is an exact solution to provide a ground truth, whereas this is not generally available for other PDEs. 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.

True solutions. Our work is guided by the vision that RL policies that are trained on small representative features with known analytic solutions have the potential to be deployed on large and complex problems without known solutions. As such, we defined a collection of parameterized function classes (true solutions) from which we sample ground truth functions $f\colon [0,1]^2 \mapsto \mathbb{R}$ with closed form for training and testing. The collection includes: *bumps*, *circles*, *steps*, and a pathological instance for existing error estimators that we call *ZZ-fail* (Figure 2). The parameters of true solutions are randomly initialized for each episode. Appendix A.1.2 contains precise definitions of each function class. These function classes are exemplars of local features, including sharp discontinuities and smooth variations, that arise in real problems. Without analytic or ground truth solutions, the alternative would be to run a reference simulation on a highly-resolved mesh to compute the reward for training.

We use these true solutions in both static and advection experiments. In the static case, the true solution is fixed and each simulation time step is precisely an RL step in which the mesh is possibly modified. In the advection case, the true solution is transported through the periodic domain and the ratio of simulation time steps to RL steps is set such that the solution returns to its original position after 10 RL steps. 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 case.

5.2. Experiments and performance metric

We conducted the following experiments for both static and advection cases to compare RL with baselines.

$\star$ 1) **Train=test**: train and test on true solutions sampled from the same function class.

$\star$ 2) **Multitask**: train a single model on all function classes and test on each individual class.

$\star$ 3) **Static$\rightarrow$advection**: test static-trained policies on advection and compare to advection policies and baselines.

$\star$ 4) **Budget$\uparrow$**: for a given function class and policies trained with a small refinement budget B (20 on static and 10 on advection), test with $B = 50$.

$\star$ 5) **Density$\uparrow$**: for policies trained with an 8×8 initial mesh partition, test on a 16×16 initial mesh partition without changing the spatial scale of the true solutions.

$\star$ 6) **Train$\neq$test:** we do not envision that one would train and test on highly different function classes in real applications, but we performed such experiments and include results in Appendix B.

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 advection case, without any refinement, the error may increase over the course of the simulation due to the accumulation of discretization error. Hence, given a refinement policy that achieves e_{final} at the end of an episode, we define its performance as $(e_{\text{no-refine, final}} - e_{\text{final}})/e_{\text{initial}}$, where $e_{\text{no-refine, final}}$ is the error at the end of an episode without any refinement.

We tuned hyperparameters for all policy architectures by training on the multitask scenario; the procedure and chosen values are given in Appendix A.3. The chosen hyperparameter values were used for all experiments. 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 average performance metric over 100 test episodes. We ensure that whenever the errors of multiple methods are compared during a test episode, that episode is initialized with the same true solution for all methods.

5.3. Implementation and baselines

We describe the high-level implementation here and provide complete details in Appendix A.2. All policy architectures use a convolutional neural network with the same architecture as their first 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. They act on each observation o^i of each input state s as described in Section 4.1. An efficient implementation for computation on a batch of states, each with varying number of observations, is described in Appendix A.2.

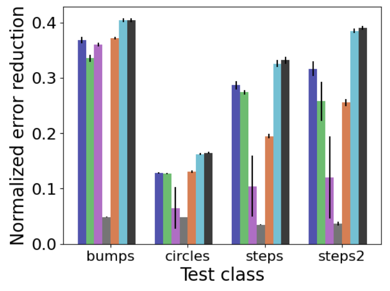
For the **Graphnet policy**, implemented with the Graph Nets library (Battaglia et al., 2018), each input state consists of node observation tensors, all edge vectors, and the adjacency matrix. Node tensors are first passed through an Independent block, after which multiple Interaction networks (Battaglia et al., 2016) act on both node and edge embeddings to produce a probability at each node (see Section 4.3).

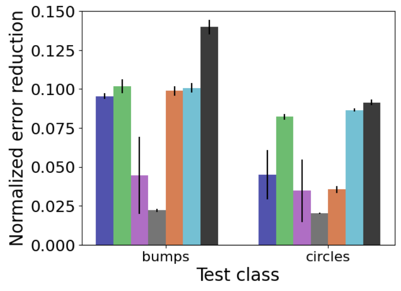
The **Hypernet policy** is parameterized by matrices $U \in \mathbb{R}^{d\times h_1}$, $V \in \mathbb{R}^{h_1\times(d\times h)}$, and $y \in \mathbb{R}^{d,h}$. U and V act on input state s to produce the main policy weights $W \in \mathbb{R}^{d\times h}$ (see Section 4.2) while y acts on s to produce a bias $b \in \mathbb{R}^h$, so that the main policy's first hidden layer is $\text{ReLU}(sW + b)$. Output probabilities are computed in the same way as IPN.

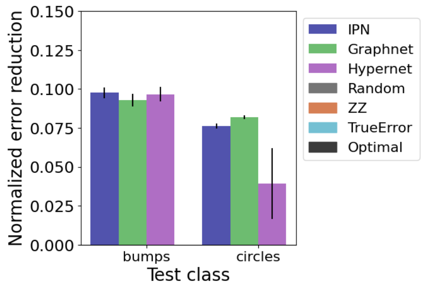
Baselines. The **ZZ** policy uses the Zienkiewicz-Zhu error estimator (Zienkiewicz & Zhu, 1992) and refines the element with the largest estimated error. The **TrueError** policy refines the element where the error of the numerical solution with respect to the true solution is largest. It is not the theoretical upper bound on performance because refining the element with largest error does not necessarily result in largest reduction of error. 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 at the next step. In contrast to RL policies, TrueError cannot be deployed at test time on systems without known solutions, while GreedyOptimal is intractable for real applications.

6. Results

We find that the proposed methods achieve performance that is competitive with baselines and, more importantly, generalize well to increased refinement budgets and initial mesh densities, and transfer effectively from a static problem to a time-dependent problem. Videos of policies and baselines on advection can be viewed at `https://sites.google.com/view/icml2021-amr`.


(a) Static


(b) Advection



(c) Static→advection

Figure 3: **Train=test** and **Static→advection**. Performance of IPN, Graphnet and Hypernetwork policies versus baselines (see Figure 14 for *ZZ-fail*). 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 function class. Performance in (c) can be directly compared with baselines in (b).

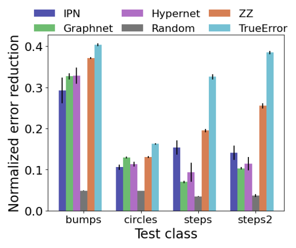
6.1. Static functions

Train=test (Figure 3a). 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 2d). 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, while Hypernet performed poorly (albeit still better than random) on all classes except *bumps*. 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 our limited hyperparameter search or the inherent difficulty of training the highly nonlinear model. All RL policies achieved non-zero error reduction on *ZZ-fail* (Figures 2c and 14) because exploration at train time and policy stochasticity at test time enabled discovery (and hence refinement) of elements that contain non-trivial features.

Multitask (Figure 4a). In the multitask training regime, we see the expected decrease in performance compared to dedicated policies trained separately on each function class. While we did not find an RL policy that significantly outperforms ZZ, our results may improved by equipping the proposed policy architectures with a stronger base RL algorithm than the simple policy gradient in this work (e.g., PPO (Schulman et al., 2017)) or building upon dedicated multitask methods (Teh et al., 2017).

6.2. Advection

Train=test (Figure 3b). 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


(a) Static

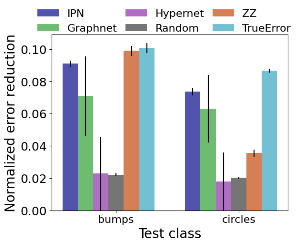
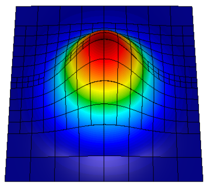

(b) Advection

Figure 4: **Multitask**: Each policy was trained on all classes.

IPN is comparable to ZZ on both functions. Hypernet is comparable to ZZ on *circles* but has high variance across independent runs. Figure 5 reveals that Graphnet spends its refinement budget more uniformly on a region of broader width along the advection path and only reaches refinement depth $= 2$ along a narrow segment travelled by the function peak, whereas TrueError reaches refinement depth $= 2$ along a wider segment.


(a) Graphnet final mesh

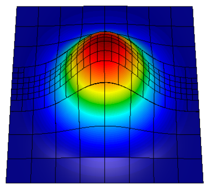

(b) TrueError final mesh

Figure 5: Graphnet trained with $B = 20$ outperforms TrueError on advection with $B = 50$

Multitask (Figure 4b). IPN and Graphnet policies trained on both *bumps* and *circles* outperformed ZZ on *circles* and IPN is close to ZZ on *bumps* (Figure 4b). The fact that both functions have smooth features and shapes is a likely reason for the positive performance of a multitask policy.

6.3. Generalization

Static→advection (Figure 3c). 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 3b vs. Figure 3c). Figure 6 shows that a static-trained policy on *bumps* with $B = 10$ correctly refines the region of propagation when deployed on *advection-bumps* with $B = 50$.

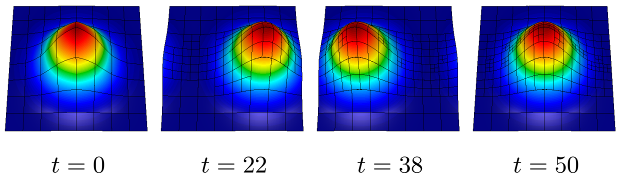


Figure 6: **Static→advection** and **Budget↑**: IPN trained on static bumps ($B = 10$) transfers to advection ($B = 50$).

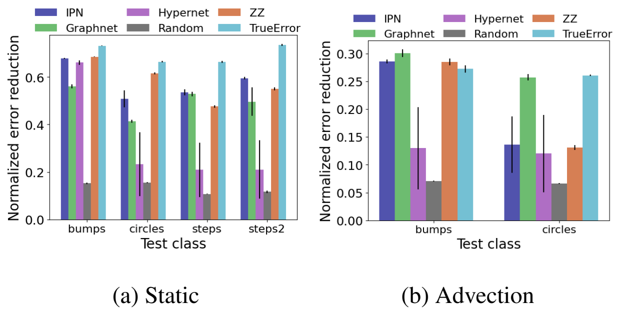


Figure 7: **Budget↑**. Policies trained with budget $B = 10$ (static) and $B = 20$ (advection) are tested with $B = 50$.

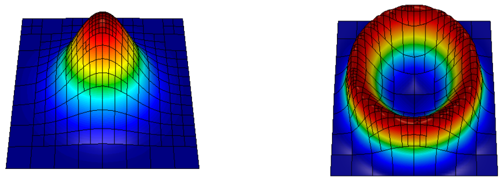


Figure 8: IPN trained with refinement budget $B = 10$ generalizes to test case with $B = 100$.

Budget↑ (Figure 7). RL policies trained with low refinement budget generalize to test cases with higher budget. In the static case, comparing Figure 3a ($B = 10$) with Figure 7a ($B = 50$) shows that the performance of RL policies relative to ZZ is generally preserved by the increase in refinement budget. Figures 8 and 11 show that an IPN trained with $B = 10$ makes qualitatively correct refinement decisions when allowed $B = 100$ during test. In the advection case (Figure 7b), 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 10 shows that an IPN trained with $B = 20$ correctly allocates a higher budget $B = 100$ to the limited region of propagation.

Density↑ (Figure 9). In the static case, the relative performance of RL policies that were trained with an 8×8 initial mesh (Figure 3a) is generally preserved when deployed on a 16×16 initial mesh (Figure 9a). 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 12 shows that IPN makes qualitatively correct refinements. On advection, relative performance is preserved on *circles* while IPN and Graphnet deproved slightly on *bumps*. (Figure 3b vs. Figure 9b)

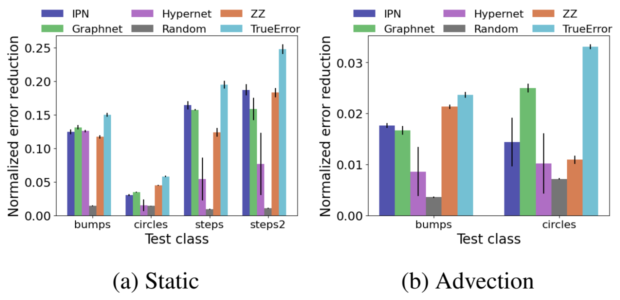


Figure 9: **Density↑**. Policies trained with initial 8×8 mesh were tested on initial 16×16 mesh.

7. Conclusion and future work

We present a novel formulation of adaptive mesh refinement as a Markov decision process with variable size state-action sets and proposed new policy architectures for scalable application of reinforcement learning. Our experiments on static and time dependent problems demonstrate that RL policies can outperform a policy based on the widely-used ZZ error estimator. For time dependent problems, we demonstrated that in some cases an RL policy can even outperform a policy that uses the exact true error at each time step to make refinement decisions, suggesting that RL can potentially provide efficiency gains beyond the reach of existing AMR approaches. Additionally we demonstrated that these RL policies can generalize to different refinement budgets and initial mesh sizes, and transfer from static to time-dependent settings. Moreover, because our RL approach does not require problem-specific knowledge or domain expertise, our results provide a path for learning novel AMR strategies for cases such as hp-refinement that currently lack effective AMR solutions. These results encourage further research at the intersection of RL and scientific computing.

Future work can build on our methods to tackle the limitations of this paper. One may consider different objectives, such as minimizing cumulative error or objectives arising from PDE-constrained optimization. One can include *derefinement* actions by generalizing our policy architectures to support a 1:2 correspondence between observation and action sets. Sampling multiple elements to refine at each time step is another open topic. It is also interesting to consider a multi-agent perspective, whereby each element is viewed as an agent and acts concurrently with all other agents.

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. Lawrence Livermore National Security, LLC. LLNL-CONF-820002.

References

Alet, F., Jeewajee, A. K., Bauza, M., Rodriguez, A., Lozano-Perez, T., and Kaelbling, L. P. Graph element networks: adaptive, structured computation and memory. *arXiv preprint arXiv:1904.09019*, 2019.

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. MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*, 81:42 – 74, 2021. ISSN 0898-1221. Development and Application of Open-source Software for Problems with Numerical PDEs.

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

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

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

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. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

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

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

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

Bohn, J. and Feischl, M. Recurrent neural networks as optimal mesh refinement strategies. *arXiv preprint arXiv:1909.04275*, 2019.

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

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

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

Burstedde, C. Adaptive mesh refinement and adjoint methods in geophysics simulations. In *EGU General Assembly Conference Abstracts*, pp. 11841, April 2013.

Červený, J., Dobrev, V., and Kolev, T. Nonconforming mesh refinement for high-order finite elements. *SIAM Journal on Scientific Computing*, 41(4):C367–C392, January 2019. doi: 10.1137/18m1193992.

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

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

Diaconis, P. Bayesian numerical analysis. *Statistical decision theory and related topics IV*, 1:163–175, 1988.

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

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

Farquhar, G., Gustafson, L., Lin, Z., Whiteson, S., Usunier, N., and Synnaeve, G. Growing action spaces. In *International Conference on Machine Learning*, pp. 3040–3051. PMLR, 2020.

Fujimoto, K. Multi-scale kinetic simulation of magnetic reconnection with dynamically adaptive meshes. *Frontiers in Physics*, 6:119, 2018. ISSN 2296-424X.

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

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

Hsieh, J.-T., Zhao, S., Eismann, S., Mirabella, L., and Ermon, S. Learning neural PDE solvers with convergence guarantees. *arXiv preprint arXiv:1906.01200*, 2019.

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

Luz, I., Galun, M., Maron, H., Basri, R., and Yavneh, I. Learning algebraic multigrid using graph neural networks. *arXiv preprint arXiv:2003.05744*, 2020.

MFEM. MFEM: Modular finite element methods [Software]. https://mfem.org, 2020.

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. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529, 2015.

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

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

Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., and Battaglia, P. W. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

Poincaré, H. *Calcul des probabilités*, volume 1. Gauthier-Villars, 1912.

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

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

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

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

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

Snaith, O. N., Park, C., Kim, J., and Rosdahl, J. Resolution convergence in cosmological hydrodynamical simulations using adaptive mesh refinement. *Monthly Notices of the Royal Astronomical Society*, 477(1):983–1003, Mar 2018. ISSN 1365-2966.

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

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

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

Teh, Y. W., Bapst, V., Czarnecki, W. M., Quan, J., Kirkpatrick, J., Hadsell, R., Heess, N., and Pascanu, R. Distral: Robust multitask reinforcement learning. In *NIPS*, 2017.

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

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

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

Zhang, Z., Wang, Y., Jimack, P. K., and Wang, H. MeshingNet: A new mesh generation method based on deep learning. *arXiv preprint arXiv:2004.07016*, 2020.

Zienkiewicz, O., Zhu, J., and Gong, N. 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, 1989.

Zienkiewicz, O. C. and Taylor, R. L. *The finite element method for solid and structural mechanics*. Elsevier, 2005.

Zienkiewicz, O. C. and Zhu, J. Z. 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, 1992.

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 all experiments except for the generalization experiment on increasing initial mesh density, where $n_x \times n_y = 16 \times 16$. 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 of the mesh is fixed by the parameter $d_{\max}$ such that the maximally-refined mesh consists of $2^{d_{\max}} n_x \times 2^{d_{\max}} n_y$ elements. For static cases, $d_{\max}$ was set to 3 whereas the time-dependent cases used $d_{\max} = 2$ 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.

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	$\{1, \dots, 6\}$
Bumps (advection)	c_x	[0.3, 0.7]
	c_y	[0.3, 0.7]
	w	[0.005, 0.05]
	n	$\{1, \dots, 4\}$
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	$\{1, \dots, 6\}$
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	$\{1, \dots, 4\}$
Steps and Steps2	o	[0, 1.0]
	θ	[0, $\pi/2$]
	n	$\{1, \dots, 6\}$

A.1.2. Ground truth functions

Bumps

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

Circles

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

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

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

ZZ-fail

$$\begin{aligned} w_x &:= \text{sgn}(\sin 32\pi x) \\ w_y &:= \text{sgn}(\sin 32\pi y) \\ f(x, y) &= \begin{cases} w_x w_y & x < 0.5 \\ 0 & \text{else} \end{cases} \end{aligned}$$

A.2. Implementation

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

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 for both static and advection problems. Chosen values of tuned hyperparameters are given in Table 2; all other experimental parameters have the same values for all methods and are listed below. We conducted tuning on the "multienv" task, in which 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 experimental parameters. 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 $(2, 2)$. We trained for 20k episodes on static problems and 10k episodes on advection problems. Each episode is initialized with refinement budget B, where $B = 10$ for static problems and $B = 20$ for advection.

A.4. 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. Policy decision times are shown in Table 3.

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}

Table 3: Mean (standard error) time in milliseconds per refinement decision on 8×8 and 16×16 initial mesh partitions.

	8×8	16×16	24×24
IPN	3.22 (0.07)	5.85 (0.05)	9.64 (0.28)
Graphnet	7.74 (0.33)	13.9 (0.43)	23.7 (0.19)
Hypernet	8.08 (0.08)	10.7 (0.05)	14.3 (0.17)
ZZ	1.96 (0.01)	6.94 (0.01)	15.5 (0.05)

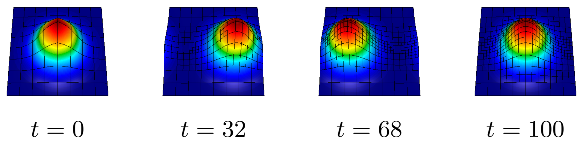


Figure 10: Advection of a bump function. RL policy trained with budget $B = 20$ generalizes to $B = 100$.

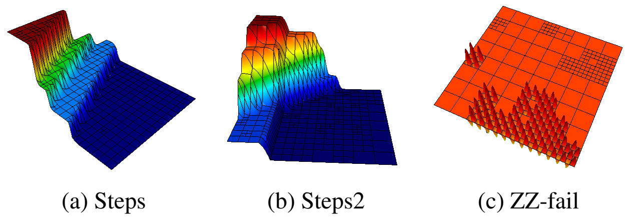


Figure 11: Generalization of policies trained with refinement budget $B = 10$ to test case with $B = 100$.

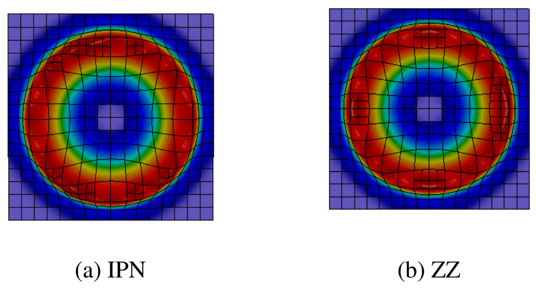


Figure 12: IPN trained on 8×8 initial mesh underperformed ZZ when tested on 16×16 initial mesh but makes qualitatively correct refinements.

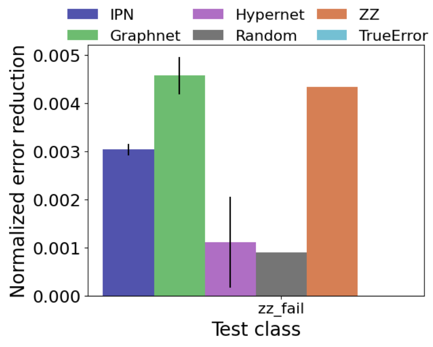
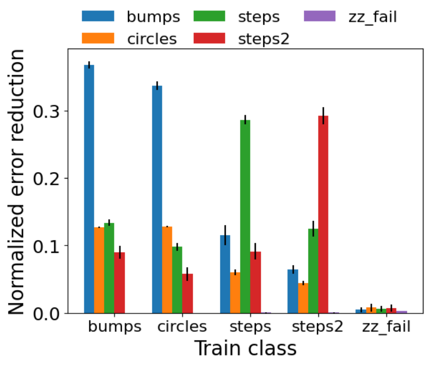


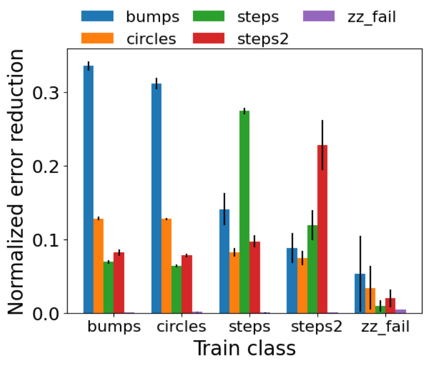
Figure 14: RL policies can reduce error in *ZZ-fail*.

B. Additional results

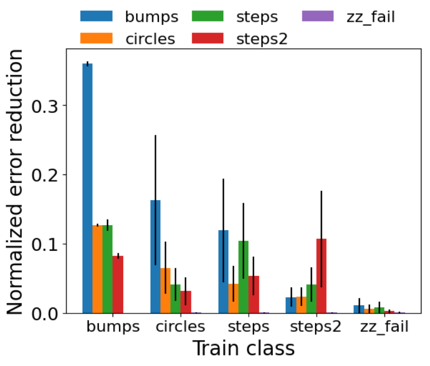
Train≠test. In the static case (Figures 13a to 13c), IPN policies trained on *circles* transfer well to *bumps* (and vice versa). Hypernet policies performed poorly overall even in the case of **train=test**, and consequently does not show comparable performance when transferring across function classes. In the advection case (Figures 13d to 13f), both IPN and Graphnet policies trained on *bumps* significantly outperformed ZZ when tested on *circles* (compare to ZZ in Figure 3b).



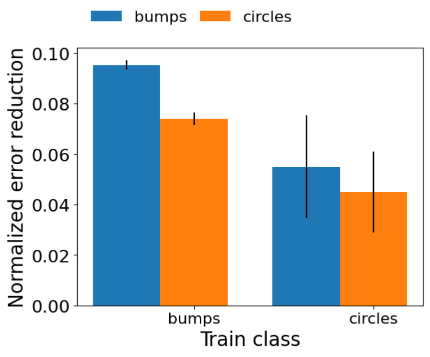
(a) Static IPN



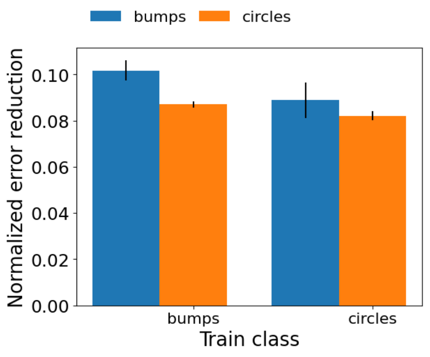
(b) Static Graphnet



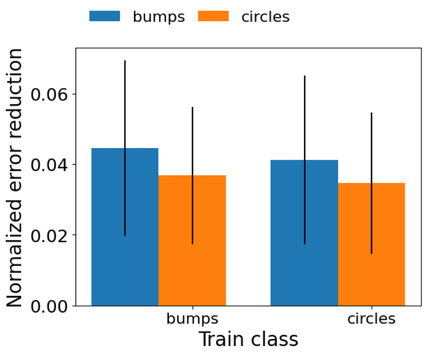
(c) Static Hypernet



(d) Advection IPN



(e) Advection Graphnet



(f) Advection Hypernet

Figure 13: 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.