Journal of Computational Physics 491 (2023) 112381

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Deep reinforcement learning for adaptive mesh refinement

Corbin Foucart, Aaron Charous, Pierre F.J. Lermusiaux [*]

Department of Mechanical Engineering, Center for Computational Science and Engineering, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, 02139, MA, USA

A R T I C L E I N F O

Article history:
Received 25 September 2022
Received in revised form 3 June 2023
Accepted 14 July 2023
Available online 22 July 2023

Keywords:
Reinforcement learning
Adaptive mesh refinement
Finite element methods
Computational fluid dynamics
Ocean modeling
Atmospheric forecasting

A B S T R A C T

Finite element discretizations of problems in computational physics often rely on adaptive mesh refinement (AMR) to preferentially resolve regions containing important features during simulation. However, these spatial refinement strategies are often heuristic and rely on domain-specific knowledge or trial-and-error. We treat the process of adaptive mesh refinement as a local, sequential decision-making problem under incomplete information, formulating AMR as a partially observable Markov decision process. Using a deep reinforcement learning (RL) approach, we train policy networks for AMR strategy directly from numerical simulation. The training process does not require an exact solution or a high-fidelity ground truth to the partial differential equation (PDE) at hand, nor does it require a pre-computed training dataset. The local nature of our deep RL (DRL) allows the policy network to be trained inexpensively on much smaller problems than those on which they are deployed. The new DRL-AMR method is not specific to any particular PDE, problem dimension, or numerical discretization. The RL policy networks, trained on simple examples, can generalize to more complex problems, and can flexibly incorporate diverse problem physics. To that end, we apply the method to a range of PDEs, using a variety of high-order discontinuous Galerkin and hybridizable discontinuous Galerkin finite element discretizations. We show that the resultant DRL policies are competitive with common AMR heuristics and strike a favorable balance between accuracy and cost such that they often lead to a higher accuracy per problem degree of freedom, and are effective across a wide class of PDEs and problems.

1. Introduction

In recent decades, the finite element community has developed principled, efficient techniques for solving partial differential equations (PDEs). Not only are these techniques extremely general methods that may be applied to almost any PDE, they also provide guarantees such as stability, consistency, and convergence [1,2]. On the other hand, the machine learning community has developed a broad set of methods to learn latent patterns from large datasets in the absence of a model [3–5]. However, for problems in computational physics, it is often the case that the PDE is an excellent model of the underlying physical phenomena, often down to the molecular level, where the continuum assumption begins to lose validity. Attempts to use machine learning to learn solutions to PDEs directly have shown promise for relatively trivial problems, but are as of yet subject to several failure modes in terms of generalization to even moderately more complicated problems [6]. Rather than ignoring the extensive body of work in either field, we propose combining techniques from numerical math-

[*] Corresponding author.
E-mail addresses: foucartc@mit.edu (C. Foucart), pierrel@mit.edu (P.F.J. Lermusiaux).

https://doi.org/10.1016/j.jcp.2023.112381
0021-9991/© 2023 Elsevier Inc. All rights reserved.

ematics and machine learning in order to preserve mathematically hard-earned guarantees while improving accuracy and efficiency by applying machine learning to the peripheral aspects of numerical methods which lack a model and rely purely on heuristics. Adaptive mesh refinement is one such aspect.

Uniform meshes are often computationally inefficient for finite element simulations in that the mesh density required to resolve complex physical features such as steep gradients or small-scale processes is used everywhere over the computational domain, even in regions where the numerical solution is smooth and much coarser resolution could be used. In many problems, such features are dynamic: eddies, meandering jets, or nonlinearly evolving cracks constitute some examples. To optimize efficiency, adaptive mesh refinement (AMR) techniques are a class of methods that dynamically modify the computational mesh during simulation in an attempt to increase resolution specifically where it is needed [7].

Most AMR techniques follow an iterative procedure of numerically solving the PDE, estimating the error on each element, marking a subset of the mesh elements for refinement or de-refinement (coarsening), and executing the alterations to the mesh [8,9]; this well-established paradigm is commonly referred to as the SOLVE→ESTIMATE→MARK→REFINE loop, terminology we will use in the present work. Following each numerical solve, the ESTIMATE process involves estimating the discretization error on each cell in the mesh. In this paper, we draw the distinction between an *error estimator*, which provides objective measures of error in a specific norm, and an *error indicator*, which offers an empirical indication as to the local magnitude of numerical error but provides no theoretical guarantees. The MARK process selects cells for coarsening and refinement based on the estimates of the error. Two common approaches are bulk refinement and fixed-number refinement [8,10–15]. The former marks all the cells responsible for a specified percentage of the (estimated) total error for refinement and similarly for coarsening. The latter refines and coarsens a fixed percentage of the total number of cells.

While AMR methodologies have allowed computational scientists to solve problems which are completely intractable on a uniform mesh [16], the application of AMR strategies remains largely heuristic, and best practices are not universally agreed upon [7]. The process of estimating the error on each cell is often complex, dependent on both the PDE and the numerical method used to solve it, and constitutes an active subject of research [17–19]. In general, for nonlinear partial differential equations, it is often difficult, and in some cases, impossible to provide an upper bound on the error. Even in the situations where rigorous error estimation is possible, the bounds on the error may not be tight, or may apply to a limited subset of problems, rendering the estimator ineffectual. As a consequence, in practice, these estimators tend to be optimistically applied as *ad hoc* error indicators. A relevant example is the well-known Kelly error "estimator" [20], which is derived from analysis specifically for the Poisson equation, but is widely employed in AMR strategies for the spatial discretizations of many other PDEs [21–25] despite its lack of theoretical applicability. Equally important, but often overlooked in the literature, the MARK process is also fraught with challenges. In attempting to refine only the cells which constitute a certain percentage of the error, bulk refinement strategies can be expensive in cases with very few cells at singularities, miss important features of the solution, and present difficulties controlling the number of cells in the mesh [12,15]. With fixed-number refinement, the number of cells in the mesh can be readily controlled, but the approach can wastefully refine too many cells [12]. Furthermore, the parameters in either of these methods (*i.e.*, the algorithmic realization of a refinement strategy with regard to the percentages of the estimated error to refine and coarsen in the case of bulk refinement or exactly how many cells to refine and coarsen in the case of fixed number refinement) are choices typically made *a priori* before a numerical simulation begins. In doing so, the balance between coarsening and refinement is implicitly assumed to be static in time, rather than dynamically driven by the behavior of the underlying solution and available computational resources.

As a result, even after decades of research, in practice, AMR strategies remain largely unprincipled and often require domain-specific knowledge, trial and error, or manual intervention. Selecting and combining an efficient set of AMR heuristics for a new problem is a challenging endeavor and an open research problem in general [12]—there exists a clear need for an automated, flexible, and principled approach to AMR, motivating the present research.

We propose the treatment of AMR as a partially observable Markov decision process (POMDP) that can be interpreted as a local Markov decision process (MDP) on each element and apply a deep reinforcement learning (RL) approach [26] in which we train an agent to increase or decrease mesh resolution, balancing improved accuracy against the computational cost associated with each mesh modification decision. Deep reinforcement learning replaces the expected reward function from the well-known classical Q-learning algorithm with a neural network as a function approximator [27], allowing for the discovery of arbitrarily sophisticated decision-making policies based on unstructured input data [28,29] and obviating the need to manually engineer aspects of the strategy. In doing so, we replace the ESTIMATE and MARK processes in AMR with a trained RL policy learned from numerical simulation.

In the present work, we focus primarily on high-order discontinuous Galerkin finite element methods (DG-FEM), in part due to their incredible success in modeling a wide range of phenomena in computational physics (see [2,30] and references therein) and due to their discontinuous representation of the numerical solution. It is this discontinuous representation that reliably links the smoothness of the local solution as measured by the interface jumps to local error [31] in a way that can be well-leveraged by a decision-making agent, as we develop in §2.2.2. The same property gives rise to the simple and effective non-conformity error estimator and its variants examined in the DG-FEM literature [19,32–35]. As demonstrated in what follows, the DG-FEM discretization plays an important role in the construction of the observation spaces for the reinforcement learning problem. However, in principle, our methodology could be applied with minor modifications to classical continuous Galerkin finite element methods or even other numerical methods such as finite difference or finite volume methods.

1.1. Related work and novel contributions

Optimal mesh refinement strategies have been shown to be theoretically learnable in a recurrent neural network setting [36]. The application of deep RL specifically to problems in computational physics is in its infancy, and recent work at the intersection between the disciplines can be found in [37–39]. A deep RL approach for the related problem of mesh generation is explored in [40]. The work in [41] constitutes the first attempt to formulate AMR as an RL problem and demonstrates the feasibility of the approach. The present work was developed concurrently and independently, and we formulate the deep RL problem differently. To avoid growing and shrinking action and observation spaces, our decision-making problem is inherently local rather than defined over the entire mesh; to that end, our observation space includes measures of local non-conformity of the solution. We elect not to impose a max refinement depth or provide a hard limit on the refinement budget in the action space; rather, we impose these restrictions implicitly through our reward function. Accordingly, we do not aim to maximize total error reduction; our reward function seeks to strike a tunable balance between the accuracy of the numerical solution and available computational resources.

In this work, we present novel theory and schemes that use a POMDP representation to formulate AMR as a deep RL problem under incomplete information and obtain a trained RL policy that can be thought of as a custom error indicator discovered through trial and error. To the best of our knowledge, this is the first work that includes both refinement and de-refinement actions as part of the resultant policy. We provide a very general framework for choosing an observation space that can incorporate physically relevant features of the PDE to be solved. The methodology is non-reliant on an exact solution or ground truth during model training or deployment. The agent learns a cost-effective refinement and coarsening strategy that balances improved solution accuracy with computational cost, as opposed to a hard threshold.

DG-FEM methods have been shown to be competitive in the *under-resolved* regimes of fluid flow simulation, both in the sense of stability and robustness of the schemes [42], and in their ability to transport high-frequency features over long time-integration horizons without altering the shape of the features or losing amplitude (dispersion and dissipation) [43,44]. Therefore, practically, the ability of DG-FEM schemes to capture and preserve physical features is a more important criterion of merit than the norm-measured errors often presented in academic convergence studies, (see [30], pp. 35-48). This is crucial, as it suggests that any performant AMR technique should be measured against its ability to resolve dynamical features while making efficient use of problem degrees of freedom. This motivates the specific form of our reward function, as well as informs our primary objective of this work; to obtain an AMR policy that accurately resolves features of the numerical solution and efficiently uses computational resources.

The paper is organized as follows. In §2, we introduce our RL formulation of the adaptive mesh refinement problem. The partially observable Markov decision process is described in §2.2, with the action and observation spaces detailed in §2.2.1-§2.2.2. The design and implementation of a new reward function that allows for model training without the need for exact or ground-truth solutions to the underlying problem are provided in §2.3. Procedures for training and deployment are given in §2.4, and a brief discussion of the deep learning policy architectures employed therein is specified in §2.5. The discontinuous Galerkin spatial discretizations for the numerical forward models are provided in §3 for different PDEs. In §4, we demonstrate the efficacy of the methodology on a set of numerical test cases spanning a variety of PDEs and numerical methods. We show that the resultant RL policies can outperform widely-used AMR heuristics in terms of accuracy per degree of freedom and that the policy corresponding to a single trained model generalizes well to different boundary conditions, forcing functions, and problem sizes. We offer concluding remarks in §5.

2. Deep reinforcement learning framework

Deep reinforcement learning combines reinforcement learning with deep learning using a neural network to represent the value function, policy, or model considered in a classical RL setting [45]. Similar to other deep learning approaches, the network is typically trained by optimization of a loss function over many episodes in which a strategy that maximizes the expected reward (1) is determined through trial and error. This approach is attractive, as it does not require domain-specific knowledge, nor does it require a large training data set; training data is generated experientially by “self-play” as the agent interacts with its environment.

Formulating an RL approach (§2.1) to a problem involves describing the underlying decision process and representation of the agent and the state (§2.2). We must specify a reward function (§2.3) which encodes desirable behaviors of the resultant learned policy. Lastly, we must specify the neural network architectures which are to represent the different parts of the RL problem (§2.5). The subsections that follow present our novel formulation of adaptive mesh refinement as a deep RL problem.

2.1. Notation

Reinforcement learning is characterized by a decision-making agent that interacts with an environment described by a state S. We consider the discrete-time case: at time step $t \in \{0, 1, 2, \dots\}$, the observable state of the environment is S_t and the action selected by the agent is A_t, the latter of which gives rise to the reward R_{t+1} and modified state S_{t+1}.

A partially observable Markov decision process (POMDP) is a formalism for describing a sequential decision-making process under incomplete information [46]. A POMDP is characterized by $(\mathcal{S}, \mathcal{A}, \mathcal{R}, T, \mathcal{O}, O, \gamma)$, where the sets $\mathcal{S}$, $\mathcal{A}$, and

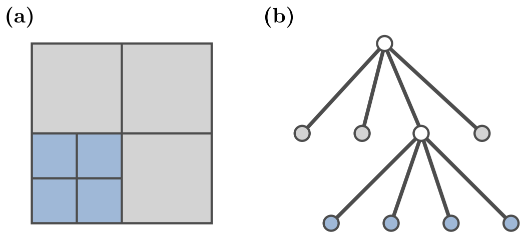

Fig. 1. (a) Adaptively refined elements in a computational mesh $\mathcal{T}_h$ generated from a single element. (b) Underlying data structure representation as a tree, with inactive parent elements and active elements as the leaves of the tree. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

$\mathcal{R}$ are the sets of possible states, actions, and rewards, respectively. The state transition function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ encodes the distribution of transition probabilities from a state $S_t = s$ to $S_{t+1} = s'$ given the action a; that is, $T(s, a, s')$ returns the probability of transitioning to state $S_{t+1} = s'$ under the action $A_t = a$ executed in state $S_t = s$. The set $\mathcal{O}$ is the set of possible observations, and the probability distribution $O(o|s, a)$ describes the probabilities over the set of possible new observations $O_{t+1} = o \in \mathcal{O}$ upon taking action $A_t = a$ in state $S_t = s$. The scalar $\gamma \in [0, 1]$ is a time discount factor—a lower discount factor motivates the agent to favor taking actions early. A POMDP differs from a standard Markov decision process (MDP) in that the agent does not have access to the complete state S_t, but rather, indirect observations of it.

The goal of RL is to find a stochastic policy function $\pi : \mathcal{O} \to \mathcal{A}$ that maps the observation space, which we take as a subset of the entire space $\mathcal{S}$, to the set of actions that maximizes the expected reward

$$Q_t(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s,\ A_t = a \right] \tag{1}$$

over an infinite time horizon [47].

2.1.1. Mesh description

We now provide the notation for the computational mesh and finite-element operations (§3). We let $\mathcal{T}_h = \cup_i K_i$ be a finite collection of non-overlapping elements K_i that discretizes the entire problem domain $\Omega \subset \mathbb{R}^d$. We refer to the boundary of the problem domain as Γ. The set $\partial\mathcal{T}_h = \{\partial K : K \in \mathcal{T}_h\}$ refers to all boundary edges and interfaces of the elements, where ∂K is the boundary of element K. For two elements K^+ and K^- sharing an edge, we define $e = \partial K^+ \cap \partial K^-$ as the edge between elements K^+ and K^-. Each edge can be classified as belonging to either ε° or ε^∂, the set of interior and boundary edges, respectively, with $\varepsilon = \varepsilon^\circ \cup \varepsilon^\partial$.

The elements K^+ and K^- have outward pointing unit normals $\boldsymbol{n}^+$ and $\boldsymbol{n}^-$, respectively. The quantities $a^\pm$ denote the traces of a on the edge e from the interior of $K^\pm$. When relevant for element-wise operations, we take as convention that the element K^- refers to the local element, and K^+ to the neighboring element. The jump $[\![\cdot]\!]$ operator for scalar quantities are then defined as $[\![a]\!] = a^- - a^+$ on the interior faces $e \in \varepsilon^\circ$. On the edges described by $\partial\mathcal{T}_h$, we can uniquely define the normal vector $\boldsymbol{n}$ as outward for a given cell and inward for its neighbor.

2.2. Elemental refinement as a local Markov decision process

As per the approach described in [10,48] and references therein, the computational mesh is composed of linear, quadrilateral, or hexahedral elements, and represented by a tree data structure. Upon refinement of an element, 2^d child elements are generated by bisection and marked as active, while the parent element is marked as inactive. The active cells of the computational mesh at any given time are the leaves of the tree. This process is schematically illustrated in Fig. 1. In this paper, we refer to elements (active or inactive) which share the same parent element as “sibling” elements.

The state S_t is characterized by the current computational mesh $\mathcal{T}_h$, its numerical solution, u_h, and any data known to the PDE (boundary conditions, forcing functions, *etc.*), which we refer to as $\mathcal{D}$. Although the POMDP would be well-defined if the observation space $\mathcal{O}$ is the set of all possible mesh configurations and numerical solutions, and even some nonlinear functions of these variables, such choices of the global state and observation space would be computationally intractable, as the agent could encounter a combinatorially growing action space to explore.

In light of this fact, we consider a local description of the state as the POMDP observation space used to train the agent. During training, the agent observes a single cell locally, takes action on the current cell alone, and receives a reward which is the effect of its local action on the entire computational domain. Defining the POMDP observation space in this way, such that $\mathcal{O} \subseteq \mathcal{S}$, allows for a fixed action space size as the size of the observation space is known and fixed. We can then formally define the current state S_t as the union of the observation space on every element of the mesh; $S_t = \cup_{K \in \mathcal{T}_h} O_K(t)$. Following every action, the agent is subsequently moved to another cell in the mesh randomly, according to a distribution $B_t(O_t, S_{t+1})$ which encodes the POMDP belief in the state of the environment and which cell should be visited next. In this work, for simplicity, we take the distribution to be uniform over all cells in the mesh. However, a Bayesian treatment of the belief state is also possible within our RL formulation.

Although it may seem counter-intuitive to train the agent on small subsets of the entire state space, formulating the learning problem locally in this way has several advantages. Considering the entire state of the mesh as an MDP leads to a growing and shrinking action space as the mesh changes, with a combinatorially growing number of action possibilities; this can be addressed [41]—however, the complicated action space may present training and generalization difficulties. Second, many problems in computational physics are governed by PDEs with strong spatial locality, which is why the linear systems arising from the discretization of PDEs are typically sparse. Therefore, viewing the RL problem on a single element rather than on the entire mesh makes sense, as the numerical solution is typically more sensitive to local phenomena. This is not always the case; *e.g.*, in elliptic PDEs, errors affect the numerical solution globally [49–51]. To address this, we specifically include elliptic PDE test cases in our numerical experiments in §4. Lastly, the resultant trained neural network constituting the RL policy has the intuitive interpretation of a custom cell-wise error indicator learned during training. With this interpretation, it can be seen that the same trained model can be deployed repeatedly over the entire mesh, in parallel if desired.

2.2.1. Action space

The action space of the agent is the discrete set $\mathcal{A} = \{\texttt{coarsen}, \texttt{do nothing}, \texttt{refine}\}$, with respect to the current cell. The goal of the RL problem can be stated succinctly as: train the agent such that it refines in areas that are locally under-resolved, coarsens in areas that are locally over-resolved, and takes no action otherwise. Furthermore, the agent should take these actions flexibly and efficiently with respect to the available computational resources.

While it is always possible to refine, the mesh may be in a state such that it is not topologically possible to coarsen the current cell. This is because the current cell could be at the coarsest possible level of the mesh, *e.g.*, a computational mesh comprised of a single cell, or because the siblings of the current element could have arbitrarily many children, and coarsening is only possible when all the sibling cells are at the highest possible refinement level (*i.e.*, leaves on the tree). For example, in the mesh depicted in Fig. 1, all of the elements marked as blue can be coarsened, whereas those marked as gray can not. Our implementation is such that if coarsening is not possible, the agent defaults to doing nothing.

2.2.2. Observation space

The observation space $\mathcal{O}$ is the observable portion of the total state $\mathcal{S}$. The observation space on an element $K \in \mathcal{T}_h$ of the computational mesh consists of:

A. The averaged absolute jump in the numerical solution over the boundary of the cell, denoted

$$\Xi_K = \frac{1}{|\partial K|} \int\limits_{\partial K} \left| [\![u_h]\!] \right| \, d\partial K$$

as well as $\Xi_{K'}$ for the cell neighbors K'

B. The average integrated jump across all mesh elements $\sum_K \Xi_K / N_K$

C. The current usage of available computational resources p

D. Any physically relevant features of the numerical solution or data $\phi(u_h, \mathcal{D})$ local to the cell, where $\phi(\cdot)$ refers to a feature and $\mathcal{D}$ refers to available data

The rationale for the inclusion of item (A.) is that discontinuities present at the element interfaces measure the non-conformity of the numerical solution. This is due to the assumption that, in the absence of a physical discontinuity, the exact solution is continuous across element interfaces. This observation is borne out by straightforward analysis, which shows a strong relationship between the residuals of the numerical solution u_h over the element interior and the jumps across the inter-element boundaries [31], leading to the development of so-called “non-conformity” error estimators [35] specific to discontinuous Galerkin finite element methods.

Although formulating the observation space of the POMDP as a completely local problem is computationally attractive, it is clear that in the absence of global information, the agent should always perform the `refine` action. Items (B.) and (C.) both serve the purpose of communicating global information regarding the relative utility of local refinement to the local decision process.

The inclusion of the average integrated jump over all mesh elements provides information about the relative estimate of the error on the current cell as compared to other cells in the mesh. During training, this element of the observation space allows the agent to decide when to forgo the opportunity to refine the current cell in order to spend computational resources elsewhere in the mesh, where the numerical errors may be greater than those observed locally.

The scalar $p \in [0, 1]$ indicates the current usage of available computational resources. In this work, we take p to be the fraction of active mesh elements out of a user-specified maximum number of elements. However, p need not have this representation: in the case of an HPC application, the value of p could be measured directly by monitoring CPU or memory usage in real time. Alternatively, p could be defined with respect to a maximum allowable wall-clock time per solution time step or could be computed from some other *a priori* allocation of computational resources such as RAM or arithmetic

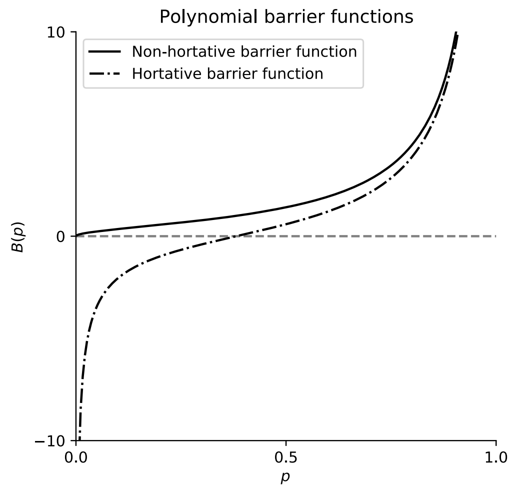

Fig. 2. Example of the non-hortative polynomial barrier functions $B(p) = \sqrt{p}/(1-p)$ and hortative barrier function $B(p) = p/(1-p) - [1/\sqrt{p} - 1]$.

throughput. Regardless of how the computational usage p is defined, its inclusion in the observation space serves to indicate to the agent the availability of computational resources; the reward function §2.3 specifies the cost-benefit relationship of making use of them.

Due to item (D.), this observation space is very flexible and general. This is by design, as it allows the user to include physically or computationally relevant information which may help the agent learn a more effective strategy. As an example, the advection problem as described in §3.2 may include the local convective velocity in the observation space, as it determines the flow of information.

Remark. In this paper, we emphasize discontinuous Galerkin methods due to the simplicity and effectiveness of the non-conformity of the local solution as a proxy for the local error. In the case of classical continuous Galerkin finite element schemes, the jump of the solution along the element boundary is identically zero, due to the continuous nature of the approximation spaces in which the numerical solution is sought [1]. However, item (A.) can be readily replaced with other indicators of non-conformity in order to generalize the methodology to other finite element discretizations. For example, the jump of the solution gradient along the element boundary can be applied as a simple surrogate in the continuous Galerkin case—the core principle is to select a numerical quantity that vanishes as the numerical solution approaches the exact solution of the PDE. However, more sophisticated indicators of non-conformity exist [18].

2.3. Designing the reward function

The reward function is an environment-provided reinforcement signal that determines the immediate reward or penalty due to each decision on the part of the agent [27]. In that sense, the reward function is central to the ultimate behavior of the agent after training and should encode all the information pertaining to what we wish our agent to accomplish. For AMR, the reward function used to train the agent should strike a balance between improvements in accuracy and incurring additional computational costs. With these conditions, our proposed reward function takes the general form of

$$[\text{accuracy}] - \gamma_c[\text{cost}]B(p). \tag{2}$$

The coefficient γ_c is a scaling factor expressing the relative importance of the accuracy of the solution versus increasing computational cost in the RL agent's policy; it can either be tuned as a hyperparameter or empirically computed for different computational regimes. The function $B(p) : [0, 1) \to [0, \infty)$ acts as an asymptotic barrier discouraging the agent from exceeding the limit of its computational resources. Two choices are shown in Fig. 2. The barrier function can be modified to encourage the agent to use a certain percentage of resources; for example, to incentivize aggressive refinement in the under-resolved case, which we refer to as a barrier function "with hortation". In this paper, unless otherwise specified, we use the non-hortative barrier function $B(p) = \sqrt{p}/(1-p)$.

It remains to define a metric for improvement in solution and change in computational cost for each of the terms in equation (2).

Accuracy. The optimality properties of the numerical solution arising from continuous and discontinuous Galerkin finite element discretizations ensure that the error of the solution (measured in an appropriate norm) decreases or remains the same upon any refinement of the computational mesh [1,2,31]. These properties are crucial, as they guarantee that any change in the error of the numerical solution, $e(u_h) = \|u_h - u_{\text{exact}}\|_{L^2(\Omega)}$, that arise as a result of refinement will be

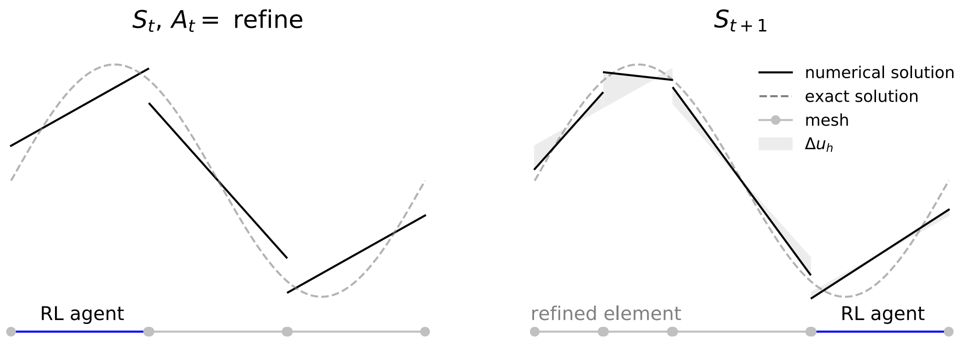

Fig. 3. Schematic of Δu_h upon refinement. The RL agent's new location is sampled from the belief distribution $B_{t_{\mathrm{RL}}}(O_{t_{\mathrm{RL}}}, S_{t_{\mathrm{RL}}+1})$.

monotonically non-increasing. Therefore the fundamental idea is to reward any change in the numerical solution upon refinement, and to penalize any change to the numerical solution upon coarsening. We define the quantity

$$\Delta u_h = \sum_{K\in\mathcal{T}_h} \int_K |u_h^{t_{\mathrm{RL}}+1} - u_h^{t_{\mathrm{RL}}}|\, dK \tag{3}$$

to represent the change in the solution, upon the action A_t, as is schematically illustrated in Fig. 3. From here onward, we will use the integers t_{RL} and $t_{\mathrm{RL}}+1$ to refer to a RL time step corresponding to the POMDP notation in §2.1 and use t to refer to time in the case of a time-dependent PDE. The quantity Δu_h represents the global difference in the numerical solution as a result of different successive levels of local refinement. The computation of Δu_h is always performed by first interpolating the coarser of the two solutions onto the finer of the two grids, then performing the integration. This way, the change in solution Δu_h upon refining and coarsening the same element are identical in magnitude, but with opposite signs, making every decision by the agent to coarsen or refine reversible.

Naively applying the reward $+\Delta u_h$ upon refinement and $-\Delta u_h$ upon coarsening is logical; however, this choice often results in an inconsistent reward signal due to scaling. One of the attractive features of high-order finite element methods is that the error in the numerical solution decreases quickly with respect to the mesh element size for problems with smooth solutions, typically as $1/h^{p_{\mathrm{order}}}$ where h is a representative size of a mesh element, and p_{order} is the polynomial order of the scheme [1,2]. As a result, the error in numerical solutions often spans many orders of magnitude over a small number of mesh refinement cycles. To account for these large differences in the scale of the error, and to prevent the initial refinements from dominating the total achievable reward during training, we scale the rewards logarithmically, awarding the agent

$$R_{\Delta u} = \pm[\log(\Delta u_h + \epsilon_{\mathrm{machine}}) - \log \epsilon_{\mathrm{machine}}],$$

where the positive and negative signs apply to the cases of refinement and coarsening, respectively. Here, $\epsilon_{\mathrm{machine}}$ is a representation of machine precision, which we take to be 10^{-16}, and is included for the edge cases in which refinement or coarsening does not change the numerical solution. Of course, it can also be a desired accuracy for the numerical solution. Lastly, the additive factor is chosen to center the positive and negative rewards around zero, rather than $\log(\epsilon_{\mathrm{machine}})$ for interpretability, and follows from our specific choice of machine precision.

Cost. We can specify the reward due to the change in computational cost upon action $a_{t_{\mathrm{RL}}}$ as the quantity

$$R_{\Delta C} = B\left(p^{t_{\mathrm{RL}}+1}\right) - B\left(p^{t_{\mathrm{RL}}}\right). \tag{4}$$

The sign of $R_{\Delta C}$ is not explicitly dependent on the action $a_{t_{\mathrm{RL}}}$, as it purely relates to the increase or decrease in computational cost as measured by p before and after the action occurs.

The reward function. Combining the constituent components, the final explicit form of the new reward function that we employ is

$$R(s_{t_{\mathrm{RL}}}, a_{t_{\mathrm{RL}}}) = \begin{cases} +[\log(\Delta u_h + \epsilon_{\mathrm{machine}}) - \log(\epsilon_{\mathrm{machine}})] - \gamma_c R_{\Delta C}, & \text{if } a_{t_{\mathrm{RL}}} = \texttt{refine} \\ -[\log(\Delta u_h + \epsilon_{\mathrm{machine}}) - \log(\epsilon_{\mathrm{machine}})] - \gamma_c R_{\Delta C}, & \text{if } a_{t_{\mathrm{RL}}} = \texttt{coarsen} \\ 0, & \text{if } a_{t_{\mathrm{RL}}} = \texttt{do nothing}. \end{cases} \tag{5}$$

As a convention, we take $R_{t_{\mathrm{RL}}+1} = R(a_{t_{\mathrm{RL}}}, s_{t_{\mathrm{RL}}})$ to be the reward returned by the environment, given the input of action and state at time t_{RL}.

We remark that equation (3) can be naively applied to vector-valued problems as well, with u_h comprising a vector-valued state rather than a scalar-valued one. However, care should be taken in selecting this state and performing differencing between refinement levels. For example, in the case of the incompressible Navier–Stokes equations, the difference can be computed with the velocity field alone or involve the pressure. In this case, relative scaling between the fields becomes important in preserving the reward signal.

2.4. Training and model deployment

To train the model on a static problem, we begin each episode with a very coarse mesh. The agent is placed randomly on a cell in the mesh; after collecting the reward for each action, the agent is moved to a different element randomly as described in §2.2. After a large given number of actions have occurred, the episode ends, with early termination possible in the case of repeated `do nothing` actions. If at any point the agent exceeds the allotted computational budget, it receives a large negative reward and the episode terminates (see Appendix C). For time-dependent problems, the training is similar. The typical RL action/reward cycles depicted in Fig. 3 are performed within a single time step, using the previous time solution state u_h^{t-1} as the starting condition. After a random number of iterations, the current RL-iterated solution is advanced to the next time step and the process is repeated. We sample from a uniform distribution $\text{Uniform}(1, \text{max episode iterations})$ to determine the number of iterations before the solution is advanced in time. In order for experience signals to be propagated to the policy network during training, all that matters is that there are areas of under- and over-resolution in the numerical solution; as long as the time advancement allows this to occur, other valid heuristics for advancing the solution are possible.

For model deployment, the trained model considers every cell in the mesh sequentially, in the order specified by the non-conformity estimator, and makes a decision based on the local observation on that particular element. The percentage p of resources in use is updated in between elements so that the model may update its recommendation. This constitutes the marking process—once every element has been visited, the recommendations of the RL policy can be executed, concluding a single AMR "cycle" for the RL model. The number of cells allowed as a budget during model deployment can be significantly larger than that used in training. The trained policy network is agnostic to the actual maximum number of cells allocated; rather, it sees only the percentage of the available cells currently in use through the parameter p.

2.5. Policy and deep learning architectures

We now describe the particular architectures we used for our AMR agent. Our deep Q-network [52] is the simplest policy. It consists of an input layer, the size of which corresponds to the size of the observation space, two hidden-layers, each with 64 neurons, and an output layer with one neuron for each possible action. Rectified linear units (ReLUs) are chosen as our activation functions. Taking the argmax of the output yields the action we take (assuming we are using the optimal policy deterministically).

The Advantage Actor-Critic (A2C) policy [53] uses the same network architecture as the deep Q-network, but it trains two networks, one to estimate the value function (the critic) and one to determine the optimal policy (the actor). At each step, the actor chooses an action that the critic evaluates to estimate the state-value and improve the policy of the actor. Finally, the Proximal Policy Optimization (PPO) policy [54] uses the same network architecture as A2C; the main difference is that PPO uses clipping in the objective function so that policy updates remain relatively small. We compare the performance of these algorithms in §4.1.1.

2.6. Summary: reinforcement learning framework

The design of the reward function is pivotal in RL. The idea behind the reward signal (5) is illustrated in Fig. 3. When the agent uses resources to refine the solution, it accumulates a reward corresponding to how much the numerical solution changes as a result of the refinement ($R_{\Delta u}$) and is penalized according to the increased usage of its resource budget ($R_{\Delta C}$). In the limiting case, in which the solution is perfectly resolved, the agent accrues only penalization for additional refinement. Conversely, when the agent elects to coarsen the resolution, it is rewarded according to the resources it saves but pays for any change in the numerical solution. In the limiting case where the solution is perfectly resolved with more elements than necessary, the agent receives a reward only for decreasing the number of elements, as the numerical solution does not change upon coarsening. In essence, the reward function provides a signal which communicates the trade-off between accuracy and computational cost. To preserve the reward signal, the reward function logarithmically scales and machine-precision-normalizes the quantities $R_{\Delta u}$ and $R_{\Delta C}$, which can span many orders of magnitude. The hyperparameter γ_c is a tunable setting that reflects the degree of displeasure incurred upon using additional resources, effectively weighting the user priority of accuracy versus cost.

By formulating the sequential decision process locally as a POMDP, the observation space provides the link between the local solution data and its conformity to the cost-benefit trade-off of refinement in that region. Informally speaking, the observation $O_{t_{\mathrm{RL}}}$ provides a signal relating local solution conformity to that of the global solution as well as the remaining computational resources and other potentially relevant data.

We summarize the algorithm for a training time step and model deployment in Fig. 4. The notation $u_h^{t_{\mathrm{RL}}} = \mathcal{M}(S_{t_{\mathrm{RL}}})$ describes running the forward model on the mesh state $S_{t_{\mathrm{RL}}}$ to compute the corresponding solution $u_h^{t_{\mathrm{RL}}}$ by numerically solving the PDE. The trained policy network is denoted π_h (§2.2), episode iterations denote a chosen number of maximum iterations during training, and we use the Boolean `done` as a sentinel for episode termination. Lastly, we refer to the belief distribution over the mesh cells $K \in \mathcal{T}_h$ at state $S_{t_{\mathrm{RL}}}$ as $B_{t_{\mathrm{RL}}}(O_{t_{\mathrm{RL}}}, S_{t_{\mathrm{RL}}})$. For the architectures we employ (§2.5), we refer to [55] for the complete specification as to how the weights of the policy network are updated as a result of the training steps taken.

```
training_step($A_{t_{\mathrm{RL}}}$, $O_{t_{\mathrm{RL}}}$):
$K_{t_{\mathrm{RL}}}$ $\leftarrow$ cell corresponding to $O_{t_{\mathrm{RL}}}$
if $A_{t_{\mathrm{RL}}}$ = refine
  $S_{t_{\mathrm{RL}}+1}$ $\leftarrow$ refine $K_{t_{\mathrm{RL}}}$
if $A_{t_{\mathrm{RL}}}$ = coarsen
  $S_{t_{\mathrm{RL}}+1}$ $\leftarrow$ coarsen $K_{t_{\mathrm{RL}}}$
update $u_h^{t_{\mathrm{RL}}+1} = \mathcal{M}(S_{t_{\mathrm{RL}}+1})$,   $p_{t_{\mathrm{RL}}+1}$
compute $\Delta u_h$ (on finer mesh) via (3), $R_{\Delta C}$ via (4)
compute $R_{t_{\mathrm{RL}}+1}$ via (5)
sample cell $K_{t_{\mathrm{RL}}+1} \leftarrow B_{t_{\mathrm{RL}}}(O_{t_{\mathrm{RL}}}, S_{t_{\mathrm{RL}}+1})$
$O_{t_{\mathrm{RL}}+1}$ $\leftarrow$ compute observation space for $K_{t_{\mathrm{RL}}+1}$
if $p_{t_{\mathrm{RL}}+1} > 1$
  $R_{t_{\mathrm{RL}}+1}$ $\leftarrow$ large, negative reward
  done $\leftarrow$ true
if iteration $\geq$ episode iterations
  done $\leftarrow$ true
return $R_{t_{\mathrm{RL}}+1}$, done, $O_{t_{\mathrm{RL}}+1}$
```

```
model_deployment($S_0$, $\pi_h$):
$S' \leftarrow S_0$
$\{T\}$ $\leftarrow$sort $K \in \mathcal{T}_h$ by $\Xi_K = \int_{\partial K} [\![u_h]\!] d\,\partial K$
For $K' \in \{T\}$
    $O_{K'}$ $\leftarrow$ compute observation space for $K'$
    $A_{K'} \leftarrow \pi_h(O_{K'})$ query policy network
    execute action $A_{K'}$, update $S'$, $p$
compute new solution $u_h' \leftarrow \mathcal{M}(S')$
return $u_h'$, $S'$
```

Fig. 4. Deep RL-AMR. Algorithms for a single RL training time step (left) and for the deployment of the trained policy over a single refinement cycle (right). The training step takes as input an action $A_{t_{\mathrm{RL}}}$ and an observation $O_{t_{\mathrm{RL}}}$, and returns the reward $R_{t_{\mathrm{RL}}+1}$, the Boolean episode termination condition done (default false), and the new observation $O_{t_{\mathrm{RL}}+1}$. The model deployment procedure takes as input a trained policy network π_h and starting state S_0; it returns the proposed next mesh state S' and the updated numerical solution u_h'.

3. Finite element spatial discretization and problem physics

3.1. Notation and approximation spaces

We define the inner products over a set $D \subset \mathbb{R}^d$ and its boundary $\partial D \subset \mathbb{R}^{d-1}$ using typical discontinuous Galerkin finite element notation as

$$(c,d)_D = \int_D cd\,\mathrm{d}D, \qquad \langle c,d\rangle_{\partial D} = \int_{\partial D} cd\,\mathrm{d}\partial D.$$

Let $\mathcal{P}^{p_{\mathrm{order}}}(D)$ denote the set of polynomials of degree p_{order} on a domain D. We consider the discontinuous finite element spaces

$$\begin{aligned} W_h^{p_{\mathrm{order}}} &= \left\{ w \in L^2(\Omega) \colon w\big|_K \in \mathcal{P}^{p_{\mathrm{order}}}(K)\ \forall K \in \mathcal{T}_h \right\}, \\ V_h^{p_{\mathrm{order}}} &= \left\{ \boldsymbol{v} \in \left[L^2(\Omega)\right]^d : \boldsymbol{v}\big|_K \in [\mathcal{P}^{p_{\mathrm{order}}}(K)]^d\ \forall K \in \mathcal{T}_h \right\}, \\ M_h^{p_{\mathrm{order}}} &= \left\{ \mu \in L^2(\varepsilon_h) : \mu\big|_e \in \mathcal{P}^{p_{\mathrm{order}}}(e)\ \forall e \in \varepsilon_h \right\}, \end{aligned}$$

where $L^2(\Omega)$ is the space of square-integrable functions on the domain Ω. Informally, $W_h^{p_{\mathrm{order}}}$ represents the space of piecewise discontinuous polynomials of degree at most p_{order} on every element in the mesh.

3.2. Linear advection equation

We will consider the linear advection equation of a tracer u

$$\begin{aligned} \frac{\partial u}{\partial t} + \nabla\cdot(\boldsymbol{c}u) &= f && \text{in } \Omega, \\ u &= g_D && \text{on } \Gamma_{\mathrm{in}}, \end{aligned} \tag{6}$$

on the domain Ω with boundary Γ separated into inflow and outflow regions $\Gamma = \Gamma_{\mathrm{in}} \cup \Gamma_{\mathrm{out}}$, that are defined according to the continuous, divergence-free advection velocity field $\boldsymbol{c}$ and outward normal $\boldsymbol{n}$,

$$\begin{aligned} \Gamma_{\mathrm{in}} &= \{x \in \Gamma \colon \boldsymbol{c}\cdot\boldsymbol{n} \leq 0\}, \\ \Gamma_{\mathrm{out}} &= \{x \in \Gamma \colon \boldsymbol{c}\cdot\boldsymbol{n} > 0\}. \end{aligned} \tag{7}$$

The linear advection problem defined in equation (6) admits the following semi-discrete discontinuous Galerkin discretization [30,31]: we seek $u_h \in W_h^{p_{\mathrm{order}}}$ such that

$$\left(w, \frac{\partial u_h}{\partial t}\right)_{\mathcal{T}_h} - (\nabla w, \boldsymbol{c} u_h)_{\mathcal{T}_h} + \left\langle [\![w]\!], u^* (\boldsymbol{c}\cdot\boldsymbol{n})\right\rangle_{\mathcal{E}^\circ} + \langle w, u_h(\boldsymbol{c}\cdot\boldsymbol{n})\rangle_{\Gamma_{\text{out}}} = (w, f)_{\mathcal{T}_h} - \langle w, g_D(\boldsymbol{c}\cdot\boldsymbol{n})\rangle_{\Gamma_{\text{in}}}, \tag{8}$$

for all $w \in W_h^{p_{\text{order}}}$. There are many choices for the single-valued, inter-element numerical flux u^*. To mimic the physics of the problem, we use the common upwinded flux

$$u^* = \begin{cases} u_h^+, & \boldsymbol{c}\cdot\boldsymbol{n} < 0 \\ u_h^-, & \boldsymbol{c}\cdot\boldsymbol{n} \geq 0. \end{cases}$$

In the steady case, the time derivative term is set to zero. In the unsteady case, a standard explicit time-integration scheme can be used for temporal discretization using a method of lines approach [2]; we use LSERK-45 [56].

3.3. Advection-diffusion equation

We will show that the methodology generalizes well across both different problem physics and different finite element discretizations. To do so, we will consider advection-diffusion PDEs, which include, as a subset, second-order Poisson-type problems. Namely, we consider the set of problems described by the PDEs

$$\begin{aligned} \frac{\partial u}{\partial t} + \nabla\cdot(\boldsymbol{c}u) - \nabla\cdot(\kappa\nabla u) &= f, && \text{in } \Omega \\ (-\kappa\nabla u + \boldsymbol{c}u)\cdot\boldsymbol{n} &= g_N, && \text{on } \Gamma_N, \\ u &= g_D, && \text{on } \Gamma_D, \end{aligned} \tag{9}$$

where Γ_D and Γ_N denote the Dirichlet and Neumann segments of the boundary, respectively. The problem in equation (9) admits a hybridizable discontinuous Galerkin (HDG) formulation, developed in [57]: we seek $(\boldsymbol{q}_h, u_h, \hat{u}_h) \in V_h^{p_{\text{order}}} \times W_h^{p_{\text{order}}} \times M_h^{p_{\text{order}}}$ such that

$$\begin{aligned} \left(\boldsymbol{v}, \kappa^{-1}\boldsymbol{q}_h\right)_{\mathcal{T}_h} - (\nabla\cdot\boldsymbol{v}, u_h)_{\mathcal{T}_h} + \left\langle \boldsymbol{v}\cdot\boldsymbol{n}, \hat{u}_h\right\rangle_{\partial\mathcal{T}_h} &= 0 \\ \left(w, \frac{\partial u_h}{\partial t}\right)_{\mathcal{T}_h} - (\nabla w, \boldsymbol{c}u_h)_{\mathcal{T}_h} + \left(w, \nabla\cdot\boldsymbol{q}_h\right)_{\mathcal{T}_h} + \left(w, (\boldsymbol{c}\cdot\boldsymbol{n})\hat{u}_h\right)_{\partial\mathcal{T}_h} + \left\langle w, \tau(u_h - \hat{u}_h)\right\rangle_{\partial\mathcal{T}_h} &= (w, f)_{\mathcal{T}_h} \\ \left\langle \mu, \boldsymbol{q}_h\cdot\boldsymbol{n} + (\boldsymbol{c}\cdot\boldsymbol{n})\hat{u}_h + \tau(u_h - \hat{u}_h)\right\rangle_{\partial\mathcal{T}_h} &= \langle \mu, g_N\rangle_{\partial\mathcal{T}_h} \end{aligned} \tag{10}$$

for all $(\boldsymbol{v}, w, \mu) \in V_h^{p_{\text{order}}} \times W_h^{p_{\text{order}}} \times M_h^{p_{\text{order}}}$. We take the diffusion coefficient $\kappa = 1$ and choose the stabilization parameter $\tau = \kappa/\ell + |\boldsymbol{c}\cdot\boldsymbol{n}|$, where we use as a diffusion length scale $\ell = 1/5$. In order to solve Poisson-like problems, we set the convective velocity field $\boldsymbol{c}$ to zero. In the unsteady case, an implicit time-integration scheme can be applied for the temporal discretization; we use the third-order backward difference formulae (BDF3). We refer the reader to [57] for a thorough discussion of these choices.

3.4. Error indicators and AMR heuristics

For the test cases in §4, we benchmark the RL policy against two common AMR heuristics, the Kelly Error indicator and a gradient indicator. The indicators are shown in Table 1.

The Kelly indicator uses parameters $c_F = n_F = h_K/24$ for element width h_K. The approximate gradient is formed using the distance vectors $y'_K = x'_K - x_K$ between the cell centers of element K and its neighbors K', as well as the matrix

$$Y = \sum_{K'} \left(\frac{y_{K'}}{\|y_{K'}\|} \frac{y_{K'}^T}{\|y_{K'}\|} \right).$$

We scale the approximate gradient in Table 1 by a power of the mesh width, using $h_K^{1+d/2}||\tilde{\nabla} u_h||$, where the integer d represents the spatial dimension of the problem. Apart from being widely used, the Kelly indicator is a natural choice of estimator to compare with the RL-agent's policy network, as it also gauges error by measuring local nonconformity—specifically, in the jump of the gradient of the solution across element faces. We avoid using the non-conformity error

Table 1
Cell-wise error indicators, contributions by face.

Error indicator	faces $F = K \cap K'$	boundary face Γ_N
Kelly	$\sum_{F\in\partial K} c_F \int_F \left[\!\left[\frac{\partial u_h}{\partial \boldsymbol{n}} \right]\!\right]^2 dF$	$n_F \int_F \left| g_N - \frac{\partial u_h}{\partial \boldsymbol{n}} \right|^2 dF$
Gradient-based	$Y^{-1} \sum_{K'} \frac{y_{K'}}{\|y_{K'}\|} \frac{u_h(x_{K'}) - u_h(x_K)}{\|y_{K'}\|}$	-

indicator itself, as it is known to exhibit undue dependence on previous refinement history and is recommended to be used in conjunction with other estimators [35]. To execute an AMR strategy based on either of these estimators, we use either a bulk or fixed-fraction rule. As described in §1, a bulk strategy refines and coarsens the cells in which the top and bottom percentages of the estimated error occurs; therefore the actual number of cells coarsened or refined is entirely dependent on the estimate. On the other hand, a fixed-fraction refinement strategy sorts the cells according to their error estimates and refines and coarsens a top and bottom percentage of the total number of cells. We denote these strategies as `bulk`(refine percentage, coarsen percentage) and, similarly, `fixed`$(\cdot, \cdot)$; *e.g.*, `fixed`(0.4, 0.6) refers to an AMR strategy where the top 40 percent of cells in terms of their estimated error are refined and the bottom 60 percent are coarsened. For additional details on both these indicators and refinement strategies, see [21]. Budget constraints can be applied to AMR heuristics as well. Imposition of a maximum depth on the tree data structure representing the mesh effectively limits the number of cells as well as the minimum cell width. Alternatively, regardless of the depth of the tree, a maximum number of cells can be imposed directly, after which no refinement is allowed to occur. A disadvantage to both these constraints is that they are simple and arithmetic in nature and make no use of the current state of the solution or percentage of resources used.

3.5. Numerical implementation

The finite element forward models were implemented in `C++` and make use of the finite element library `deal.II` [21]. The POMDP characterizing the RL environment was implemented using the OpenAI Gym framework [58], and the custom deep RL architectures were implemented in the open-source RL framework Stable-Baselines 3 (SB3) [55].

Unless otherwise specified, the policy architecture used is A2C, and we take the hyperparameter $\gamma_c = 25$ (see §2.3, §4.1), which we empirically found to give good performance over a wide range of test cases. The main DRL-AMR parameters for each numerical experiment are listed in Table 2. Additional details on RL training are given in Appendix C. As is typical in deep RL, the particular number of training time steps used is not particularly meaningful as compared to the order of magnitude of the total number of steps (see discussion in §4.1.1), as long as the policy network is given sufficient time to increase its mean episodic reward significantly from its starting performance. For all experiments, we used on the order of 10^5 RL training time steps, consisting of 100-200 time step episodes, and training took on the order of 1-3 hours on a desktop computer without GPU acceleration. More detailed benchmarking would be extraneous, as performance in terms of training time is dominated by the cost of running the numerical solver rather than updating the policy network, for all but trivial problems. Similarly, we benchmark the resultant heuristics and policy networks using L^2-error per degree of freedom as a figure of merit. Model deployment requires only a forward pass through the policy network, whereas AMR heuristics require computation of the relevant estimators; however, these two operations happen in different programming languages, making CPU-time or wall-clock time performance comparisons between the two unclear. However, as problem size grows, the cost of running the numerical solver dominates the time-to-solution. Since the underlying PDE solvers are the same for both the DRL-AMR and heuristic approaches, we avoid comparisons using CPU time or wall-clock times and show L^2-error per degree of freedom directly, as this metric succinctly illustrates the cost of obtaining a given accuracy using each approach.

All linear systems arising from the finite element discretizations are solved using direct solvers (UMFPACK) to avoid the complicating factors of iterative solver tolerances and stopping criteria. All surface and volume integral operators are discretized with Gaussian quadrature using $p_{\text{order}} + 1$ one-dimensional (1D) quadrature points in each spatial direction, where p_{order} is the polynomial order of the finite element space $W_h^{p_{\text{order}}}$.

L^2-errors, when shown, are computed as a post-processing step using a known exact solution and numerically integrated using $p_{\text{order}} + 3$ Gaussian quadrature points in each spatial direction to ensure that the calculation of errors is not adversely affected by integration error. We re-emphasize that no exact solutions are used at any point during training or deployment of the RL model itself.

4. Numerical experiments

The numerical experiments are designed to demonstrate the features and performance of the deep RL AMR (DRL-AMR) approach. We start with a 1D, steady linear advection problem in §4.1 as an illustrative example to exhibit the feasibility of the approach and to demonstratively explore the characteristics of the RL policy. We discuss details related to training and model evaluation, compare the performance of different RL algorithms, and easily interpret results due to the simplicity of the test case.

Subsequent experiments focus on the performance and generalizability of the DRL-AMR method. In §4.2, we use the same trained RL model as in §4.1, but apply it to a different linear advection problem to demonstrate generalization of the policy to different boundary conditions and forcing functions. In §4.3, to show the generalization of the method to unsteady dynamics, we train a model on a 1D time-dependent advection problem. Lastly, to support the claim that the approach is not specific to a particular PDE or finite element scheme, we apply the DRL-AMR framework to the Poisson equation discretized with an HDG scheme (§3.3). As the Poisson equation is a second-order, elliptic PDE, its solutions are characterized by different physics than the hyperbolic, first-order advection problems solved in §(4.1-4.3). Similarly, as HDG schemes are mixed methods, their formulation (10) involves vector-valued finite element spaces and use of a traced finite

Table 2
Numerical experiments of §4: Training parameters of the DRL-AMR models, AMR heuristics, and indicators.

§	p_{order}	γ_c	training episodes	training budget	AMR heuristic	indicator
4.1	3	25	$2 \cdot 10^4$	25 cells	`bulk`(0.5, 0.5)	gradient-based
4.2	3	25	$2 \cdot 10^4$	25 cells	`fixed`(0.5, 0.1)	Kelly
4.3	3	100	$5 \cdot 10^4$	25 cells	`bulk`(0.5, 0.1)	gradient-based
4.4	3	25	$2 \cdot 10^5$	20 cells	`fixed`(0.5, 0.5)	Kelly
4.5	2	25	$2 \cdot 10^5$	110 cells	`bulk`(0.5, 0.5)	gradient-based
4.6	4	25	$3 \cdot 10^5$	200 cells	`bulk`(0.5, 0.5)	Kelly
4.7	3	25	$3 \cdot 10^5$	200 cells	`bulk`(0.6, 0.4)	Kelly

element space [57] on the mesh skeleton; that is, the underlying numerical method is significantly different than those used for the solution of the advection equation (*cf.* (8)).

To show that the methods are not relegated to small or one-dimensional problems and that the DRL-AMR method generalizes to higher spatial dimensions and larger, more complex problems, we examine the performance of deep RL policies trained on 2D problems in §(4.5-4.6). The steady advection problem in §4.5 allows evaluation of DRL-AMR in the context of the two-dimensional generalization of the problem in §4.1. The steady advection-diffusion problem in §4.6 examines the effectiveness of the deep RL policy when both advection and diffusion processes occur, and highlights its ability to detect and resolve non-trivial features in an automated way by using solution smoothness. The unsteady advection problem in §4.7 shows the ability of DRL-AMR to preserve salient physical features of a numerical solution over long-time integration horizons. Throughout the entire set of numerical experiments, we focus on the goal stated in the introduction of providing a numerical solution that captures all physically relevant features efficiently in terms of problem degrees of freedom, avoiding spurious diffusion and dispersion effects due to under-resolution.

The parameters used in training the RL models for all numerical experiments are shown in Table 2, along with the AMR heuristics to which we compared the models.

4.1. Steady 1D linear advection

For proof-of-concept, we consider the linear advection equation described in §3.2 on the 1D spatial domain $\Omega = [0, 1]$. We choose the boundary conditions g_D at the inlet and forcing function f such that the exact solution takes the form

$$u(x) = 1 - \tanh\left[\alpha(1 - 4(x - 1/4))\right],$$

where we take as the steepness parameter $\alpha = 10$. As the exact solution has the form of a smooth step function, the resultant meshes and numerical solutions are easily interpretable—resolution should be concentrated in the steep gradient region around the "step". In light of this, we will use this illustrative example to demonstrate the features of the trained deep RL policy, as well as the details of training and model deployment. Subsequent sections will focus on generalizability.

Using a numerical solution of polynomial order $p_{\text{order}} = 3$, we train the RL-agent for $2 \cdot 10^4$ episodes on a computational "budget" of 25 cells (§2.3). However, at the time of model deployment, we are free to give the trained RL policy whatever budget we wish; we emphasize that this is because in general, we would like to train our model on much cheaper problems than those we intend to solve.

During deployment, starting with a very coarse mesh consisting of 4 elements, we perform 6 AMR cycles comparing the DRL-AMR model to an AMR heuristic that attempts to refine the elements responsible for the top 50 percent of the total error and attempts to coarsen the elements responsible for the bottom 50 percent of total error as measured by the approximate gradient error indicator (Table 1). That is, a `bulk`(0.5, 0.5) strategy (see §3.4). We expect this heuristic indicator to perform well, as this test case has only one feature, the steep gradient in the center of the domain.

Fig. 5 shows the numerical solution and meshes proposed by the two approaches. We see that the deep RL agent deployed with a 25-cell budget is able to find a high-quality solution with fewer elements than that of the AMR heuristic recommendation. This is corroborated by the more precise comparison in Fig. 6, which shows that at model deployments with budgets of 25 and 500 cells, the RL agent outperforms the AMR heuristic over six refinement cycles in terms of L^2-accuracy per degree of freedom. That is, the RL policy provides a numerical solution of approximately the same accuracy as that of the AMR heuristic, but does so with fewer degrees of freedom after the refinement cycles, both on the problem size it was trained on, as well as at a much larger problem size.

Although for 1D problems, differences in problem degrees of freedom are small, this example illustrates the central difference between the learned policy and the classical AMR approaches. AMR algorithms mark cells for refinement either in terms of the estimated volume fraction of total error in the case of bulk refinement or by a certain number of cells in the case of fixed-number refinement. In either case, the AMR algorithms rely on estimators purely to decide which cells to refine, but the actual refinement behavior is in some sense, specified *a priori*. Similarly, the commonly-used way to prevent AMR heuristics from continuing to refine beyond a computational limit is to manually specify a max refinement depth, again an a priori choice uninformed by the particulars of the solver or PDE, and specified everywhere. The DRL-AMR approach, in contrast, not only estimates which cells are likely responsible for a disproportionately large or small share of the total

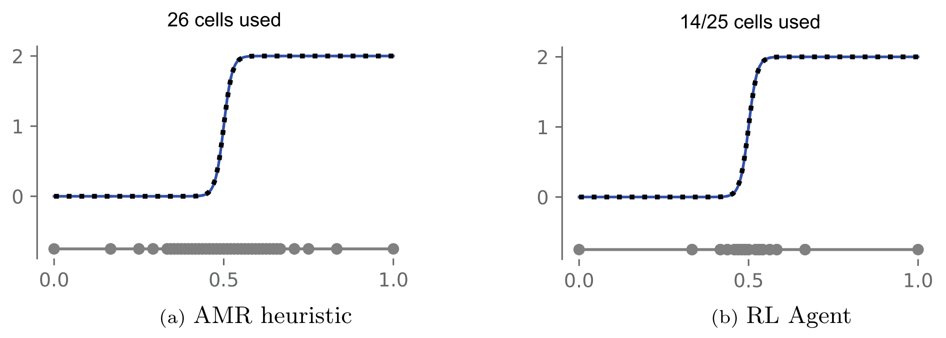

Fig. 5. Steady 1D linear advection (§4.1). Numerical solution with the mesh resulting from 6 cycles of refinement, using the approximate gradient error indicator as an AMR heuristic and the deep RL policy resulting from training. The exact solution is overlaid (dashed) in both cases for comparison.

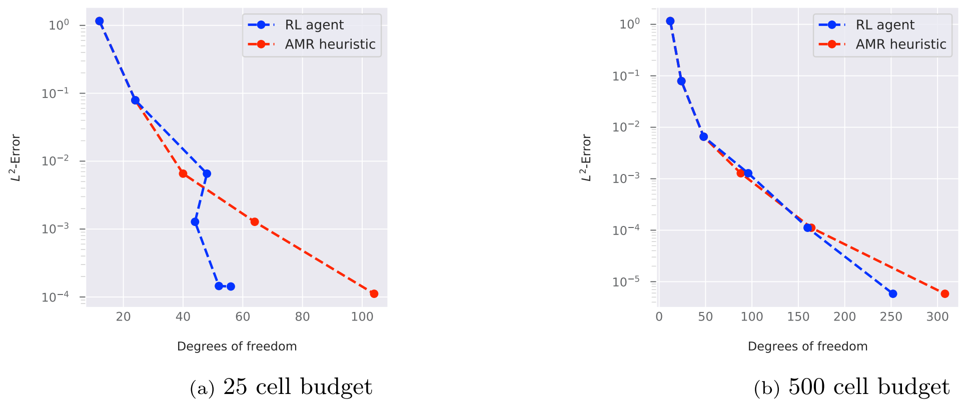

Fig. 6. Steady 1D linear advection (§4.1). L^2-error plotted against problem degrees of freedom over 6 cycles of refinement, using the gradient-based error indicator as an AMR heuristic and the deep RL policy: (6a) deployed with a computational budget of 25 elements; (6b) with a computational budget of 500 elements.

error but also estimates a stopping point, beyond which the solution smoothness indicates that additional decreases in error are likely to be marginal from the perspective of adding new unresolved features to the solution. This is because, during training, the RL agent is given global information related to available computational resources, which it balances against the expectation of change in the solution upon local refinement with respect to computational cost.

Where the stopping point occurs, in terms of the solution error, is controllable through the hyperparameter γ_c. A larger value of γ_c means a cheaper numerical solution with a greater aversion to incurring a cost. On the other hand, a small value of γ_c will result in a numerical solution with a lower error and less tolerance for interface jumps. Informally, γ_c is a user-specified setting indicating the trade-off between cost and accuracy, and is a way to inform the agent "how accurate" versus "how cheap" to make the solution. In Appendix A, we provide an approach to remove γ_c as a training hyperparameter entirely, and instead make it a user-chosen value during deployment by modifying the policy architecture; however, this discussion is beyond the scope of the main contributions of this paper, and we train using a single value of γ_c for all of the numerical experiments in this section. Next, we explore the training and deployment behavior of the RL policy for this simple case to better understand the model.

4.1.1. Training and deployment considerations

Initialization. At the start of each training episode, we can either allow the RL agent to begin on the coarsest possible mesh (coarse initialization) or initialize the mesh to a random state by performing a random number of refinements.

Random initialization and coarse initialization perform similarly for the small problem sizes on which they are trained; however, per unit training time, we find that random initialization often yields better policies when deployed on substantially larger problems. Fig. 7b compares the performance of two DRL-AMR models trained for 10^5 training time steps, one with the mesh in a coarse state at the beginning of each episode, and the other with a mesh in a random state at the beginning of each episode. Both models were trained on the linear advection problem in §4.1 with a maximum budget of 25 cells but were deployed with a budget of 500 cells over six refinement cycles. Comparing the accuracy with the problem degrees of freedom, it is clear that the model trained with random initialization significantly outperforms the model with coarse initialization. Both models begin the refinement cycle on the same coarse mesh during deployment. The model trained with random initialization finds a solution of the same accuracy but uses only half the elements as the model trained with coarse initialization. We obtained similar results for other test cases we ran.

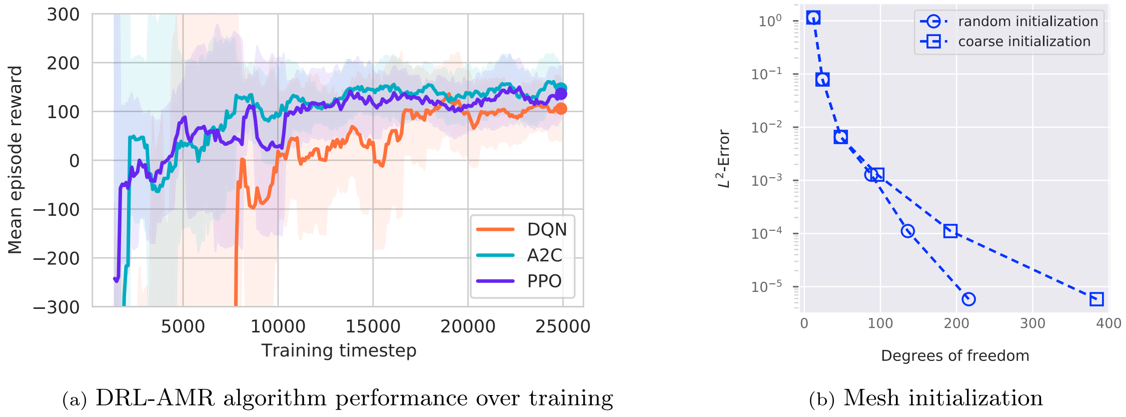

Fig. 7. For the steady linear advection test in §4.1: (a) Performance achieved during training for different deep RL training algorithms, as measured by average episode reward (solid lines). Shaded bands indicate the rolling sample standard deviation of the episodic reward over a 10 episode window. (b) Performance of DRL-AMR models trained using random state initialization and coarse initialization, as measured by L^2-error per degree of freedom over 6 refinement cycles (on a 25-cell budget example and deployed with a 500-cell budget).

We hypothesize that random initialization during training leads to a more aggressive exploration of the action space and produces better results because initializing the mesh to a random state produces regions that are over-refined and under-refined. On the other hand, initialization on a coarse mesh tends to produce an under-resolved solution everywhere, leading the agent to become biased towards refinement during deployment. As might be expected, in the limit of long training times, performance becomes comparable between the two initialization strategies.

In light of these performance advantages, all results shown correspond to models trained using random initialization.

Learning algorithms. The performance—as measured by mean episodic reward—for the three RL policy architectures (DQN, A2C, and PPO, see §2.5) are shown in Fig. 7a. The sample variance bands are found to tighten non-monotonically over the training duration, indicating more consistent performance. All three algorithms achieve similar mean reward and variance after roughly 25,000 training time steps, successfully solving the problem. However, as is common in RL problems, the model performance during training is non-monotonic, and it is advantageous to periodically save the most performant model state for deployment, rather than deploying the policy occurring at the end of the training window. The most performant model at any time during training is defined as the model with the highest mean episodic reward over a given lookback window–we use the SB3 library default of 500 training time steps. In general, we have not found any particular architecture to be consistently better in terms of performance over the set of test cases considered in this paper. We note however that we have not yet attempted to accelerate the training process through optimization of training parameters such as batch size and episode length.

Model deployment. Unlike the AMR heuristic, the deployed DRL-AMR model finds solutions by apportioning computational resources over the mesh in a way that handles the smoothness/efficiency trade-off, then changes the mesh topology over the subsequent refinement cycles in order to increase accuracy. The DRL-AMR algorithm thus considers a richer set of strategies than a purely greedy strategy.

To illustrate this, Fig. 8 shows the mesh and resultant numerical solution over a set of six refinement cycles for an RL-agent trained in the same manner as in §4.1, but with a hyperparameter value of $\gamma_c = 100$, chosen to provide coarser solutions for the ease of visualization. Although cycles 2-6 all contain the order of 10 elements, the topology of their allocation over the domain changes to concentrate around the steep gradient region. This showcases a particular strength of the method: namely, that the decision of the number of elements to use during the exploratory cycles while preserving solution smoothness is delegated to the machine learning model, rather than manually needing to be specified. This allows the number of elements to increase and decrease exploratively, as opposed to AMR heuristics, which must increase or decrease the number of elements according to the error estimator or by a fixed number.

In practice, the cost of each cycle depends on the problem size but we find that the trained RL policies reach a converged mesh state in a few cycles, typically around 5-10, independent of problem size. This is in part because the number of cells can potentially double or halve every cycle, so locally under-resolved regions either become resolved very quickly, or the computational budget is approached and refinement recommendations become more conservative.

Introspection of the neural network model. As a deep RL policy is ultimately a neural network, we can query the trained neural network in order to visualize the policy suggestions for different inputs. Here, we are interested in visualizing the DRL recommendations (refine, no-change, or coarsen) as a function of generic solution properties. The input space to a policy network is the observation space and, in our case, it includes the numerical solution over the element. To sample this network, we could for example use the solutions on all elements and marginalize over these element solutions to create a map (decision versus solution properties). However, this sampling may not be sufficiently complete (limited by the element solutions used) and solution properties would need to be defined. To avoid to these issues, as discussed in §2.2.2, we could instead change the observation space and retrain a new DRL model for which we can easily sample the entire observation

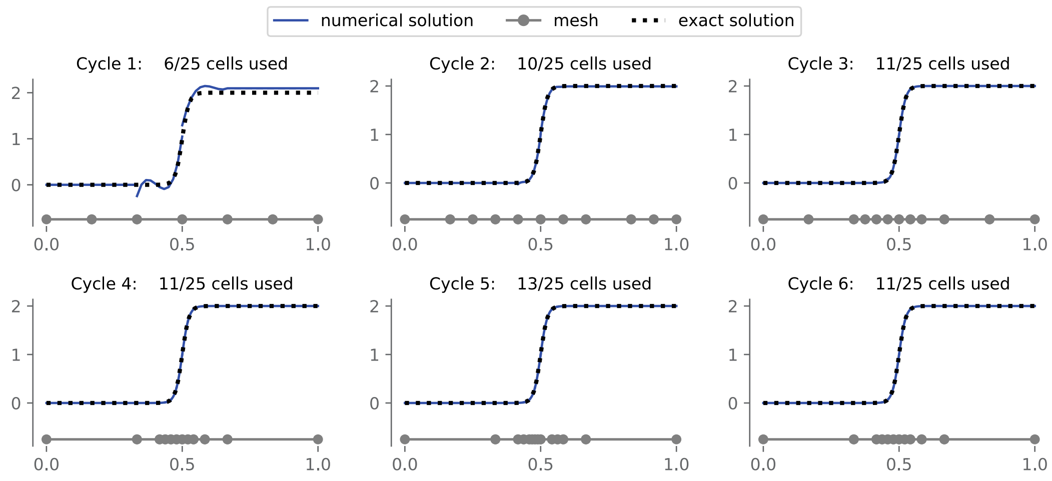

Fig. 8. Steady 1D linear advection (§4.1). State of the numerical solution and mesh during the deployment of the deep RL agent AMR policy, over six refinement cycles.

space and easily interpret results. Therefore, we train a new model using a reduced input observation space consisting only of the cell boundary jump, the mean boundary jump over all cells, and the current use of computational resources p. The purpose of this visualization is not to evaluate the performance of this simpler model but to show that such DRL-AMR networks are able to learn a sensible mapping between the observation space and the refinement recommendation.

Fig. 9 shows the recommendations of the simplified model trained on the problem in §4.1 using a value $\gamma_c = 25$. We consider a range of [-16, -1] in log space, as these are the range of jump values encountered by the network for this test case; recommendations outside of this region are uninformed by data encountered during training. Each sub-figure shows the decision boundary regions corresponding to the action recommended by the policy. At low use of computational resources ($p = 0.3$), the model suggests mostly refinement, even in regions where the observed boundary jump is significantly lower than the average over the entire mesh. We hypothesize that this corresponds to exploratory refinement in some sense. At moderate use of computational resources ($p = 0.5$), the model provides more conservative recommendations, suggesting refinement where the local boundary jump exceeds the mesh mean and is not less than 10^{-8} in terms of magnitude. When computational resources become scarce ($p = 0.7$), the model suggests coarsening except in regions where the local log boundary jump is much greater than the mean. The decision boundaries are highly nonlinear, even for this simplified observation space. This demonstrates the potential power and flexibility of the DRL-AMR approach, as it learns complex relationships between any physical relevant information included in the observation space and the utility of coarsening or refining.

4.2. Generalization to different boundary conditions and forcing functions

We deploy the same trained DRL-AMR model in §4.1 to the same PDE, but on a different domain $\Omega = (-4, 4)$, with a different set of boundary conditions and forcing. We choose the boundary conditions g_D at the inlet and forcing function f such that the exact solution takes the form

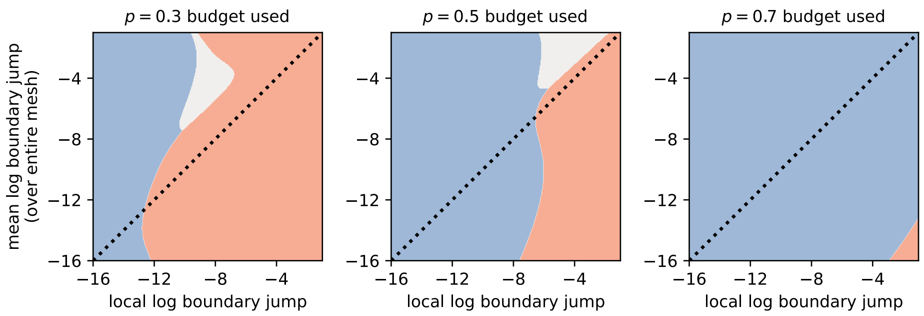

Fig. 9. Samples of the policy recommendation of a newly trained DRL-AMR network for steady 1D linear advection (§4.1), as a function of a simplified observation space consisting of the cell boundary jump, mean jump, and current use of computational resources. Decision boundaries are shown between network recommendations to coarsen (blue), do nothing (gray), and refine (salmon). The dashed line indicates where the local boundary jump is the same as the mean cell-wise boundary jump over the entire mesh.

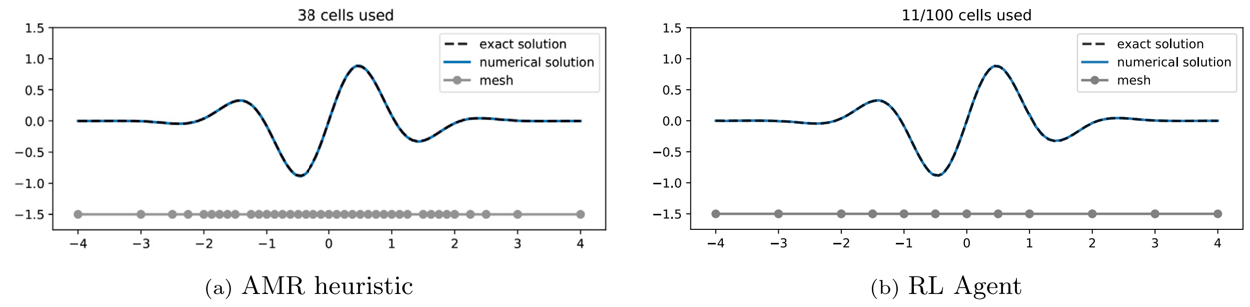

Fig. 10. DRL-AMR model trained on the test case in §4.1 but deployed on the problem in equation (11). Numerical solution with the mesh resulting from 6 cycles of refinement, using the Kelly error indicator as an AMR heuristic and the RL policy resulting from training. The exact solution is overlaid in both cases for comparison.

$$u(x) = \sin(nx)\exp\left(-\frac{1}{2}x^2\right). \tag{11}$$

At the time of model deployment, we give the trained RL policy a budget of 100 elements. We evaluate the model over 6 AMR cycles this time using an AMR heuristic which makes use of the Kelly error indicator (Table 1, [20]) and implements a `fixed`(0.5, 0.1) strategy, as described in §3.4.

The final mesh and numerical solution for both is shown in Fig. 10. Our findings are similar to the test case in the previous section. Both methods find a mesh and corresponding numerical solution that very closely matches the exact solution, however, the RL policy finds a coarse mesh on which the solution is well represented and stops refinement past the fourth refinement cycle. Similar to Fig. 6, after the six refinement cycles, the RL policy returns a mesh that provides strictly better accuracy per problem degree of freedom (not shown).

This highlights the generality of the method; the RL policy did not suggest refining close to the center of the domain as it did for the example in §4.1. This is to be expected, as the RL agent is never given global location information during training, only local cell information along with the surrounding interface jumps; therefore it's impossible to over-fit the RL agent to a specific training example during training. However, we remark that if the agent were trained against a pathological example such as the Weierstrass function, where the correct action could always be to refine, this will be reflected in the trained model, although this can hardly be considered overfitting. Because the agent learns to relate features of the PDE and local jumps to the local smoothness of the solution, we find that the agent generalizes well across different test cases from the same PDE. This is an important characteristic of the approach, as we wish to deploy the trained model on problems of interest other than the subset on which the model was trained.

4.3. Unsteady 1D linear advection

As an extension to the steady experiments, we consider the time-dependent Sommerfeld wave equation

$$\begin{aligned} \frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \qquad & \text{in } \Omega \times [0, T], \\ u(x,t) = g_D(t) \qquad & \text{on } \Gamma_{\text{in}}, \\ u(x,0) = u_0\,, & \end{aligned}$$

on the computational domain $\Omega = (-4, 4)$. The Dirichlet boundary condition $g_D(t)$ at the inlet is chosen according to the exact solution to the time-dependent wave equation $u_{\text{exact}}(x,t) = u_0(x-t)$. We consider numerical solutions at a polynomial order $p_{\text{order}} = 3$.

Gaussian pulse: training and deployment. In the first example, we use a Gaussian initial condition $u_0 = \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$. The scalar constants are $\mu = -4$ and $\sigma^2 = 0.25$, and the background velocity field is taken to be $c = 1$. The outlet boundary of the domain is taken to be an outflow condition.

We train the DRL-AMR model for $5 \cdot 10^4$ RL time steps using the time-dependent solver and training procedure as detailed in §2.4 with a computational budget of 25 elements and using a scaling factor value $\gamma_c = 100$ to render a relatively coarse numerical solution (2), as the computational cost is more heavily penalized. All other training parameters assume the default values (§3.5). For the AMR heuristic, we employ a gradient indicator (Table 1), that approximates the gradient of the numerical solution to estimate the error. Given the fast-decaying tails of the Gaussian pulse being advected, the gradient-based refinement indicator can be expected to accurately recommend coarsening outside of the pulse.

During deployment, we allow the AMR heuristic and the RL policy to perform one cycle of refinement/coarsening at every time step before advancing the numerical solution in time. We consider an increased budget of 100 elements for the RL-agent, and for the AMR heuristic we perform a bulk-refinement approach where we refine the cells responsible for the top 50 percent of the numerical error as measured by the gradient indicator, and coarsen the bottom 50 percent of the

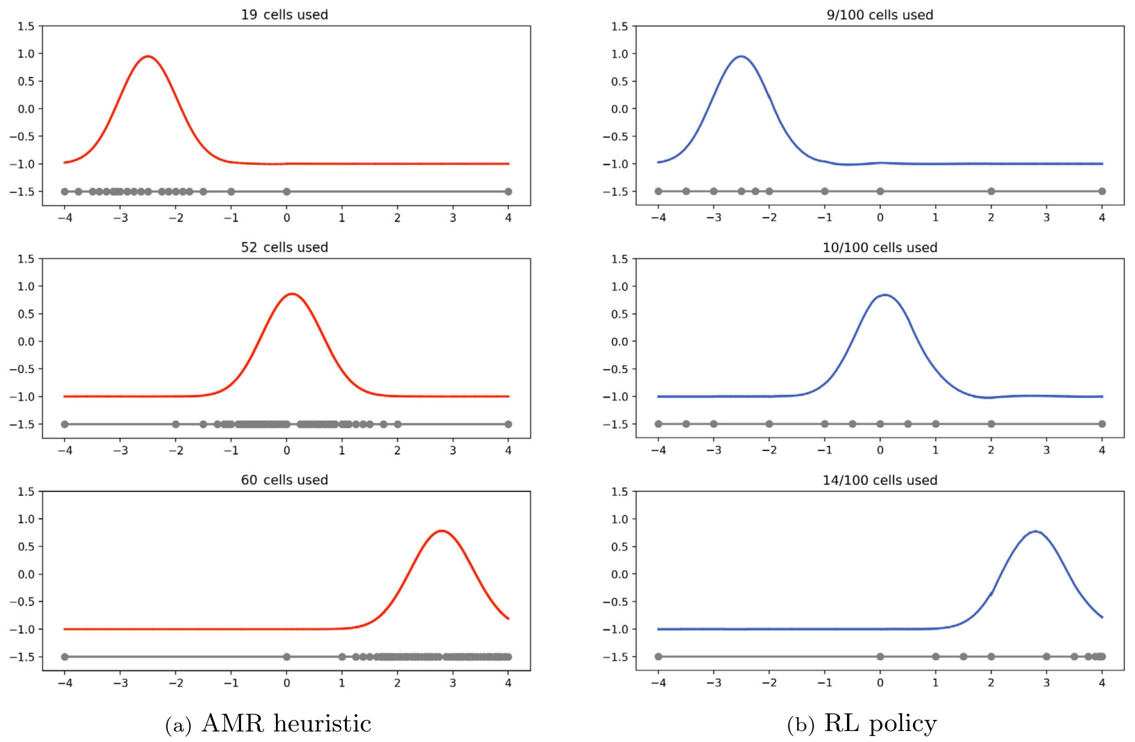

Fig. 11. Unsteady 1D linear advection: Gaussian pulse. Numerical solution with time-dependent AMR policies using a gradient-based heuristic (left column) and the trained RL policy (right column), at times $T = 1.6$ (top), $T = 4.2$ (middle), and $T = 6.9$ (bottom).

numerical error as measured by the same, *i.e.*, a `bulk`(0.5, 0.5) strategy. Using a time step $\Delta t = 0.01$, we simulate to a final time of $T = 7$. The results are shown in Fig. 11. Similar to the other experiments, we find that both algorithms are able to capture the advection of the initial condition; the RL policy does so using a computational mesh with far fewer elements than the AMR heuristic. Namely, the DRL-AMR model is able to preserve the features of the solution while using fewer than 25 percent of the computational cost incurred by the AMR heuristic. Due to the increased value of γ_c as compared to the other numerical experiments considered thus far, the RL agent is more tolerant of a lack of smoothness in the solution. This is reflected in the small discontinuities of the numerical solution at the location $x = 0$ at $t = 1.6$ and at $x = 2$ at $t = 6.9$.

Deployment on multi-feature wave advection. We showed in §4.2 that the trained policy was able to generalize beyond its training setup in terms of boundary conditions and forcing functions for the steady advection problem. To highlight the same in this time-dependent case, we use the policy network trained on the time-dependent wave equation example (Fig. 11), but deploy it on a more feature-rich initial condition

$$u_0 = \left(\alpha_1 e^{-\frac{1}{2\sigma^2}} (x - \mu) + \alpha_2 \sin(\alpha_2 x) \right) (x - 2\pi) \tag{12}$$

with parameters $\alpha_1 = 3$, $\alpha_2 = 2$, $\mu = 0$, and $\sigma = 2/25$, a different background velocity $c = \pi$, and a larger budget of 350 cells.

The domain and background velocity field are chosen as $\Omega = [-2\pi, 2\pi]$ and $c = \pi$, respectively, such that the exact solution is advected to its initial profile at the final time $T = 4$. To ensure the numerical error is primarily due to spatial discretization rather than temporal error, we use for time-marching a fourth-order explicit Runge Kutta (LSRK4) scheme and a small time step $\Delta t = 1 \cdot 10^{-3}$. For spatial discretization, we use the discontinuous Galerkin scheme of §3.2 with a polynomial order $p_{\text{order}} = 3$. We compare the performance of the RL-agent to an AMR heuristic using the Kelly error estimator and a `bulk`(0.5, 0.3) strategy.

Numerical results are shown in Fig. 12 and Fig. 13. Fig. 12a shows the initial condition and the mesh of 64 elements on which the AMR heuristic and RL agent are initialized. Fig. 12b shows the numerical solution and mesh given by the RL policy network at $t = 1$. Qualitatively, the features of the initial condition have been well-preserved, despite the use of only 27% of the active cell budget; the L^2-errors at the final time were $3.13 \cdot 10^{-2}$ and $3.21 \cdot 10^{-2}$ for the AMR heuristic and the RL-agent solution, respectively. To provide a more thorough comparison, Fig. 13 compares the numerical solutions provided by the AMR heuristic and the RL agent at the final time $T = 4$; both demonstrate agreement with the exact solution–however, the RL-agent does so using far fewer elements. The RL-agent uses only 92, fewer than half of the 210 cells used by the AMR heuristic.

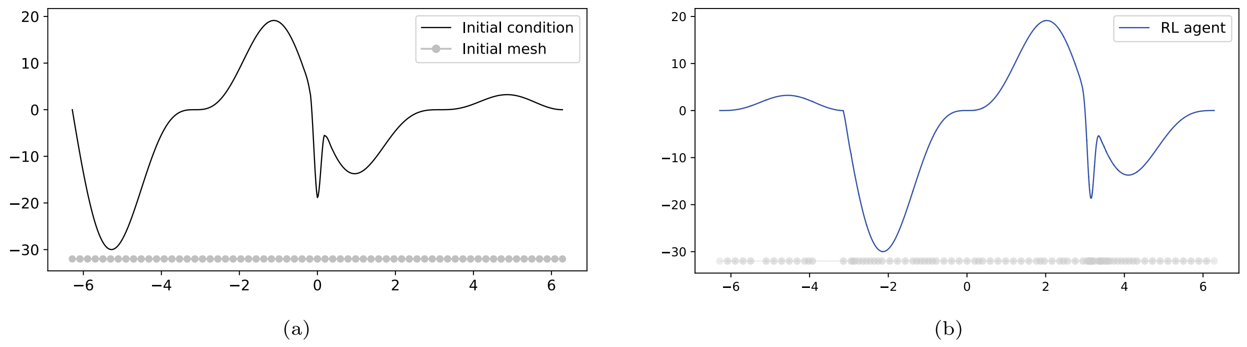

Fig. 12. Unsteady 1D linear advection: multi-feature wave. (Left) Initial condition (12) and initial mesh of 64 elements. (Right) Numerical solution at time $t = 1$, integrated using RL-agent (94 active cells) trained on the Gaussian pulse. The mesh is shown with transparency for ease of visualization.

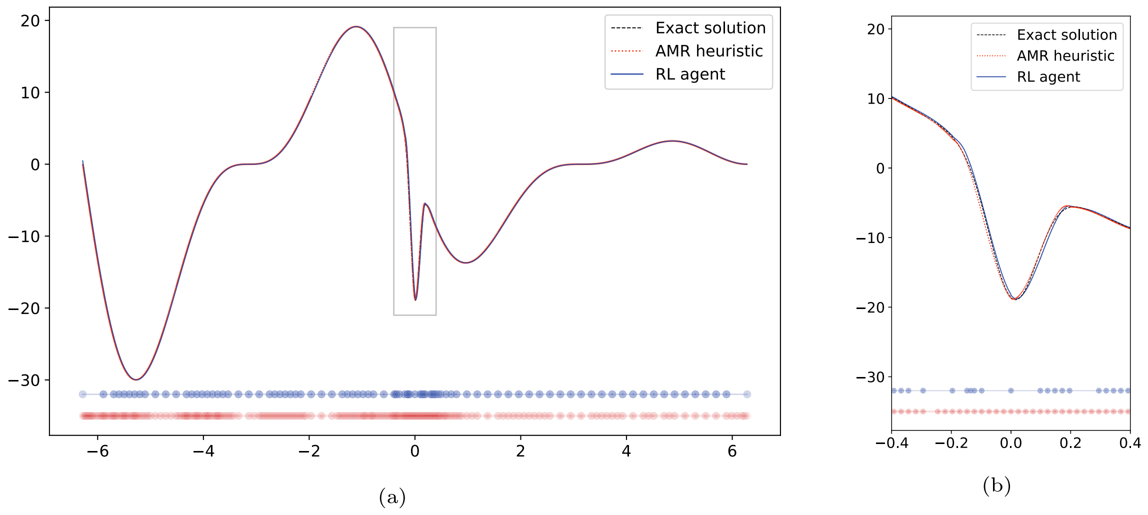

Fig. 13. Unsteady 1D linear advection: multi-feature wave. (Left) Numerical solutions and corresponding meshes at the final time $T = 4$ for the RL-agent trained on the Gaussian pulse and for the AMR heuristic, along with the exact solution. (Right) Zoomed-in center region corresponding to the box in (a).

Although both approaches appear to prioritize refinement in roughly the same regions, by examining the fast-varying feature in the center region, depicted in Fig. 13b, we see that the RL agent is more selective about refinement, even in local regions with sharp features. To explain this, we remark that the reward function is formulated such that the agent learns a map between a local observation and the expected utility of refinement; therefore, even if the agent is in an area of “interesting” activity, the numerical solution on a particular element may suggest that there’s diminishing local utility to performing refinement. Moreover, the use of neighbor cell information, as well as the convective velocity in the observation space can result in anticipatory behavior in the recommendations, whereas the AMR heuristic will not.

We emphasize that due to the local nature of the observation space, over-fitting to a particular feature encountered during the training example(s) is not something to be concerned with, as the agent never has access to the solution as a whole, and the same feature will be covered with different numbers of cells over coarsening and refinement. Ultimately, the RL agent is primarily concerned with learning the relationship between a local signature of the numerical solution and the conformity of the solution; this is why the agent performs well at preserving solution features. In that sense, the DRL-AMR approach is advantageous in that applying it should not be particular to a specific PDE or numerical scheme, a claim we investigate in the next section.

4.4. Generalization to different PDEs

To demonstrate that the DRL-AMR approach is not specific to the advection equation considered above, we apply it to the second-order Poisson equation, a subset of advection-diffusion PDEs (§3.3).

$$-\frac{\partial^2 u}{\partial x^2} = f \qquad \text{in } \Omega = (-1, 1), \tag{13}$$

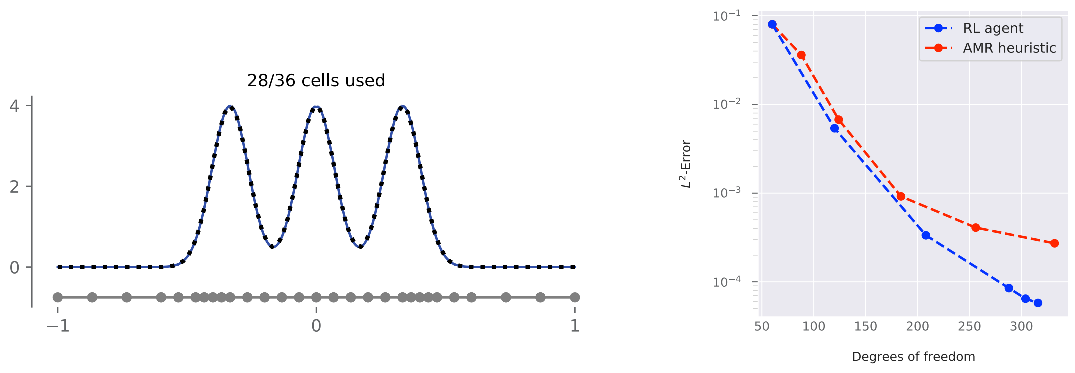

Fig. 14. Steady 1D Poisson problem (13). (Left) Mesh distribution and corresponding numerical solution resulting from 6 cycles of refinement using the trained RL policy with a budget of 36 cells; the exact solution (14) (black, dashed line) is overlaid for comparison. (Right) L^2-Error of the numerical solution vs problem degrees of freedom over 6 mesh refinement cycles for the RL agent and the Kelly error estimator as AMR heuristic.

with mixed boundary conditions consisting of a Dirichlet condition $u = g_D$ on boundary $\Gamma_D = \{-1\}$ and Neumann condition $\nabla u \cdot \boldsymbol{n} = g_N$ on boundary $\Gamma_N = \{1\}$. The boundary conditions and forcing function are deduced from the exact solution, chosen to be a superposition of Gaussian functions

$$u = \sum_{i=1}^{3} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - r_i)^2}{\sigma^2}\right) \tag{14}$$

with parameters $r_1 = -1/3$, $r_2 = 0$, $r_3 = 1/3$ and $\sigma = 1/10$. We discretize the problem using HDG finite elements as described in §3.3, with polynomial order $p_{\text{order}} = 3$. We set the diffusivity coefficient $\kappa = 1$ and the convective velocity $c = 0$.

We train an RL model for $2 \cdot 10^5$ training time steps, and all non-specified training parameters assume the default values (§3.5). Results of the trained RL policy are shown in Fig. 14. We see that the features of the solution are well resolved, and match the exact solution.

Furthermore, the trained policy identifies regions on the outer edges of the Gaussian mixture as in need of refinement; we hypothesize that this region is specifically sensitive to Gibbs phenomena caused by the change from a near-zero solution to a non-trivial one. The more precise comparison between L^2-error and problem degrees of freedom in Fig. 14 shows that the DRL-AMR model outperforms the AMR heuristic. As the Kelly error indicator was specifically designed for this type of Poisson problem and can be referred to as an error estimator in this context, the fact that the RL policy is competitive is an encouraging result.

Lastly, the PDE in (13) is elliptic. The Green's function for the Laplacian operator decays as $1/r$, where r is the euclidean distance from the point in question, in contrast to the advection equation, where errors are advected along with the solution along characteristic paths. Therefore the numerical solutions as well as effective adaptive refinement strategies can be expected to be of a different character than those for purely hyperbolic problems. The fact that the RL methodology was nonetheless able to satisfactorily solve the problem lends credence to the POMDP formulation and use of a local observation space during training.

4.5. Steady 2D linear advection

To examine the generalizability and performance in the context of larger, higher-dimensional problems, we solve the steady version of the linear advection equation (§3.2) on the two-dimensional (2D) spatial domain $\Omega = (0, 1)^2$ with a circular, counter-clockwise convective velocity field $\boldsymbol{c}(x) = 1/\|x\|_2\,(-x_2, x_1)$. On the inflow boundary Γ_{in}, we specify the boundary condition g_D according to the chosen solution,

$$u(r) = 1 - \tanh\left(\alpha(1 - 4(r - x_0))\right), \tag{15}$$

where r denotes the radius in cylindrical coordinates. We use the parameters $\alpha = 10$, $x_0 = 1/4$ and forcing function $f = 0$. The exact solution is a two-dimensional, cylindrical coordinate generalization of the test case considered in §4.1; that is, the solution contains a smooth, steep gradient. However, unlike the test case in §4.1, due to the higher problem dimensionality, the steep gradient is present both in the domain interior as well as at the domain boundary.

To train the network, we solve the linear advection problem

$$\begin{aligned} \nabla \cdot (\boldsymbol{c}u) &= 0, && \text{in } \Omega, \\ u &= u(r), && \text{in } \Gamma_{\text{in}} \end{aligned}$$

using the DG discretization given in §3.2 with elements of polynomial order $p_{\text{order}} = 2$, and deploy the model on the same problem. However, we train the RL policy network with a resource budget of only 110 cells, limiting the learning process to

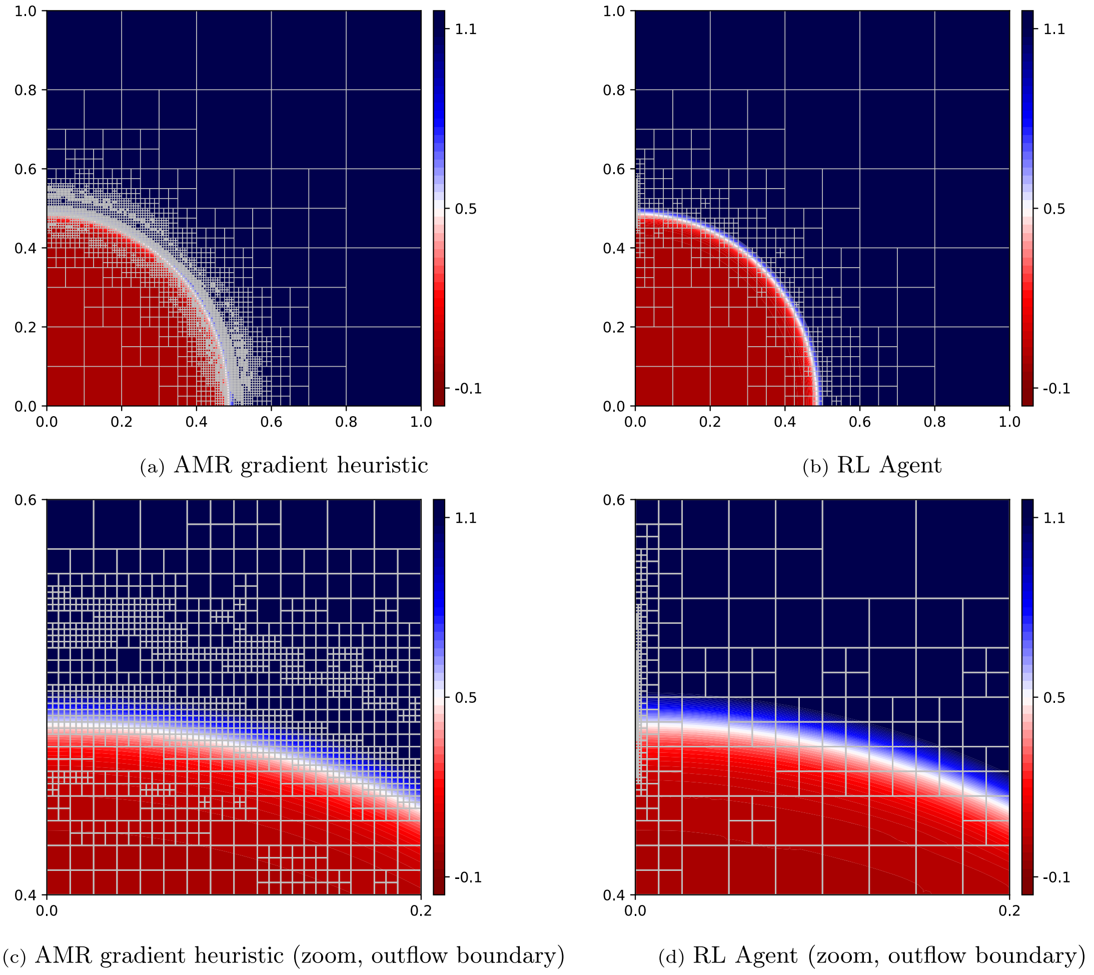

Fig. 15. Steady 2D linear advection. Numerical solution overlaid with the mesh refined using a gradient-based error indicator and the RL policy (1000 cell budget). (Top row) Entire problem domain. (Bottom row) Zoomed-in portion of the domain where the steep gradient meets the left outflow boundary.

a relatively small number of elements compared to the deployment budgets; all other training parameters are the default values (§3.5). After $2 \cdot 10^5$ training, time steps, we deploy the trained RL policy over a series of 6 refinement cycles, starting with a coarse mesh of 25 elements with 5 elements in each direction.

We compare the performance of the RL agent to the gradient-based error indicator (Table 1) that uses a `bulk`(0.5, 0.5) refinement strategy. The AMR heuristic is given an effective budget of 3200 cells by limiting the maximum refinement depth; we compare this approach to RL policies with both larger and smaller budgets. Similar to the test case considered in §4.1, this AMR heuristic can be expected to perform well, given the steep gradient as the dominant feature of the exact solution.

In Fig. 15, we show the two numerical solutions overlaid with the final meshes and find that the RL policy is able to outperform the heuristic. The RL policy was deployed with a budget of 1000 cells; it makes use of roughly 2/3 of its budget allocation by the final refinement cycle. Qualitatively, both approaches are able to resolve the steep gradient; however, compared to the AMR heuristic, the RL agent has much more conservatively refined the mesh around the slope. Instead, the RL agent refines preferentially in the region where the steep gradient encounters the outflow boundary (*cf.* Fig. 15c and Fig. 15d). In training, the RL agent learned that the numerical solution is sensitive to the discontinuities arising where the gradient meets the outflow boundary. Although the agent has no information on where cells are located, the cells in this problem region exhibited behaviors in their observation space that indicated the numerical solution would improve by refining them. That is to say, the RL policy learned a non-trivial, spatially-heterogeneous strategy for regional refinement. In contrast, the gradient-based indicator detected the steep discontinuity but was not able to exploit the sensitivity of the error to the resolution close to the outflow boundary.

In Fig. 16, we show the L^2-norm of the solution errors as a function of the number of degrees of freedom in the numerical discretization at each level of the refinement cycle, showing results for the RL policy deployed at different cell budgets, specifically 200, 1500, and 5000 elements. By the final refinement cycle, the RL policy finds a solution of comparable or

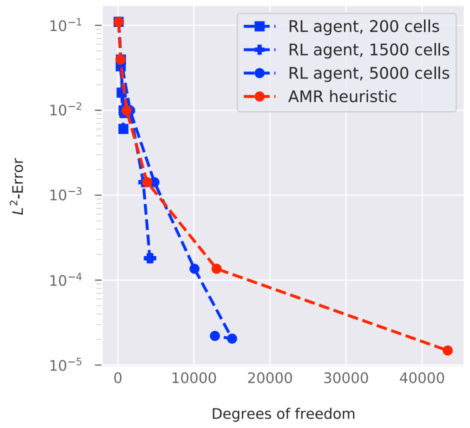

Fig. 16. Steady 2D linear advection. L^2-error as a function of the number of degrees of freedom for the refinement cycles resulting from a gradient-based error indicator and the RL policy.

better accuracy than that of the AMR heuristic for the same number of degrees of freedom. With a budget of 5000 cells, the RL agent remains well under budget but substantially outperforms the AMR heuristic. This is due to the aggressive refinement of the heuristic around the gradient region; after five refinement cycles, nearly all the mesh cells are located in this region, and an additional refinement cycle results in performance similar to a uniform refinement *i.e.*, nearly quadrupling the degrees of freedom. This poor performance of the AMR heuristic can be somewhat attributed to the nature of the problem, as discontinuous Galerkin methods for transport equations can be plagued by reduced convergence rates on arbitrary meshes, especially meshes which are not flow-aligned [59], as is the case in this example.

4.6. Steady 2D advection-diffusion equation

Section §4.5 involves a relatively simple steady numerical solution, with a single steep gradient region. It is natural to ask whether the methodology can capture solutions with more complicated features. In this section, we thus consider the steady advection-diffusion problem (9) with mixed boundary conditions, and the less straightforward hybridizable finite element discretization (10) of the same.

We choose a circular velocity field $\boldsymbol{c} = (x_2, -x_1)$ in the domain $\Omega = (-1, 1)^2$. The top and bottom boundary conditions are Dirichlet Γ_D, whereas the left and right conditions are Neumann Γ_N. The boundary values and forcing function are deduced from the exact solution, a superposition of Gaussian functions chosen to create non-trivial features. The details of the exact solution are given in Appendix B. The problem is discretized at polynomial order $p_{\text{order}} = 4$ and makes use of element-local post processing, resulting in an observed optimal convergence order of $p_{\text{order}} + 2$ upon uniform mesh refinement (see [57]).

We train the RL policy for $3 \cdot 10^5$ training time steps, using a budget of 200 cells; all other training parameters are the default values (§3.5). As in the other numerical experiments, the training budget was chosen to be significantly cheaper than the deployment budget. For comparison, we consider an AMR heuristic that makes use of the Kelly error indicator with a `bulk`(50, 50) refinement strategy, refining and coarsening cells responsible for the top 50 percent and bottom 50 percent of the total estimated error, respectively. The AMR heuristic is given an effective budget of 3200 cells by limiting the maximum refinement depth, applying a cell limit as discussed in §3.4. We compare this approach to RL policies with both larger and smaller budgets; we choose larger and smaller budget as a comparison because, as is shown in the following, the RL-policy network recommends substantially fewer cells than the AMR heuristic in either case. Both strategies are employed over 5 refinement cycles, beginning from the coarse mesh shown in Fig. 17a.

The results of the comparison between the two algorithms are given in Fig. 17. Examination of the L^2-error per degree of freedom (not shown) again demonstrates that the RL agent is competitive with any of the AMR heuristics considered in this paper. However, in this experiment, we focus on the relative ability of the heuristic and RL AMR strategies to resolve relevant features of the solution, pursuant to the discussion in §1.1. We see from Fig. 17a that on the coarse starting mesh, the numerical solution is acutely under-resolved. Both the AMR heuristic (Fig. 17b) and the trained RL agent (Figs. 17c, 17d) are able to satisfactorily resolve the features of the numerical solution, but the RL agent does so with a more parsimonious allocation of elements over the mesh. Even when the computational budget is increased from 1500 cells to 5000 cells, the RL agent makes only minor adjustments to the mesh.

The explanation for this is that the trained DRL-AMR model makes decisions based on the learned relationship between the features of the observation space local to each element, and the change in the numerical solution upon refinement. The HDG finite element schemes (10) enjoy a faster convergence rate and, typically, better overall accuracy than the advection DG scheme (8) due to its dual formulation and element-wise post-processing, as discussed above. Additionally, we use a relatively high-order (sixth-order effective convergence rate) finite element scheme as compared to the smooth step example

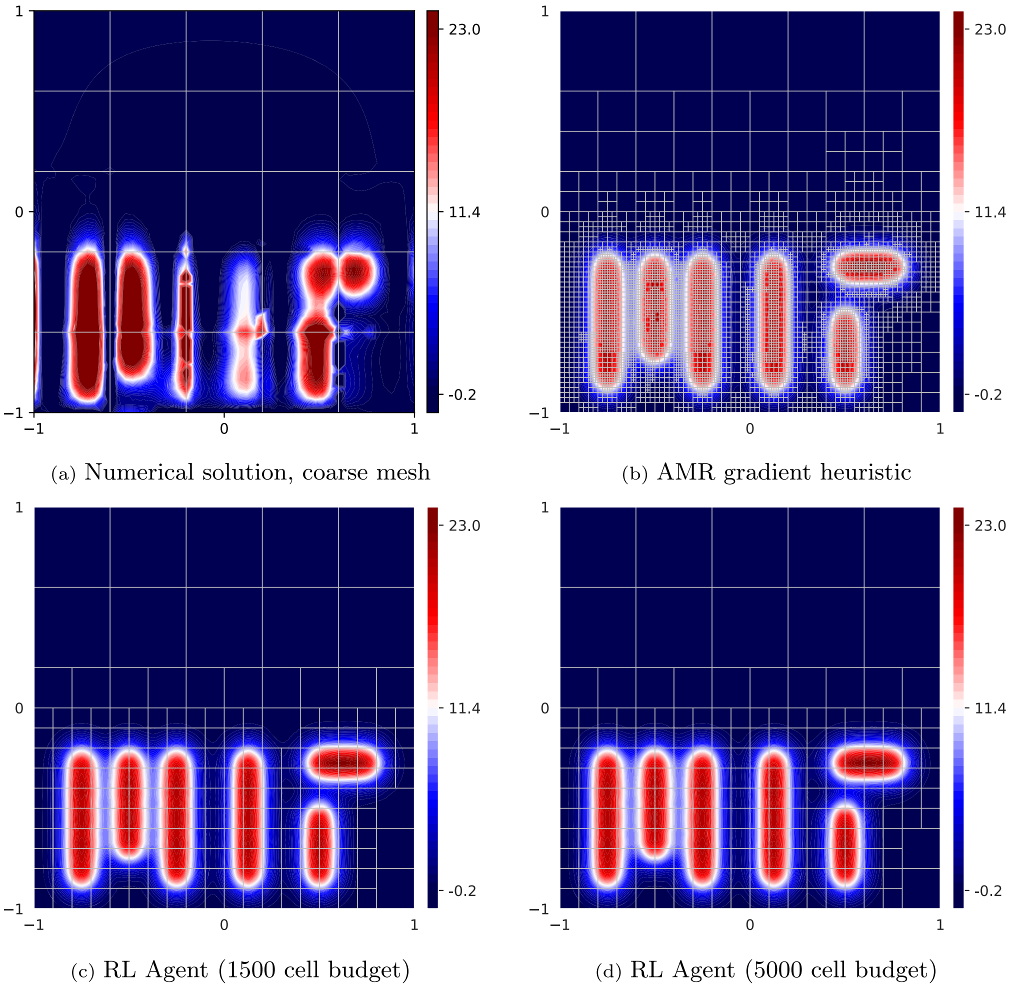

Fig. 17. Steady 2D advection-diffusion. Numerical solutions overlaid with the corresponding final mesh.

in §4.5, which exhibited second-order convergence. The RL policy is able to detect the fast convergence to the exact solution (according to the local numerical solution as well as the jump discontinuities over the boundary of each element) and elects to use only a small portion of its computational budget in order to resolve the solution, approximately 8 percent and 2 percent for budgets of 1500 cells and 5000 cells, respectively. We hypothesize that the inclusion of additional HDG-specific features into the observation space, such as the numerical gradient $\boldsymbol{q}_h$ or the approximate numerical flux on the mesh skeleton $\hat{u}_h$ (10) could additionally improve performance.

In summary, the RL policy is able to judiciously allocate resources in order to capture non-trivial features of a numerical solution. Furthermore, the generality of the RL methodology extends to more complicated finite element schemes and is able to take advantage of their properties—in this case, high-order convergence due to a post-processed solution.

4.7. Unsteady 2D linear advection

In section §4.3, we showed that the RL policy, trained on a simple unsteady example, was able to perform well when deployed on a much more complex flow, with a different setup than that seen in training. We also showed that the complex features of the solution were preserved over many time integration steps. In this section, we extend these results to two dimensions: specifically, unsteady 2D linear advection problems (§3.2).

Unsteady 2D Gaussian pulse advection: training. For DRL training, we employ the advection of a simple Gaussian as illustrated in Fig. 18, extending the Gaussian pulse of §4.3 to 2D. The initial tracer condition (Fig. 18a) is

$$u_0 = \exp\left(-\frac{1}{2\sigma^2}\left((x-\mu_x)^2 + (y-\mu_y)^2\right)\right)$$

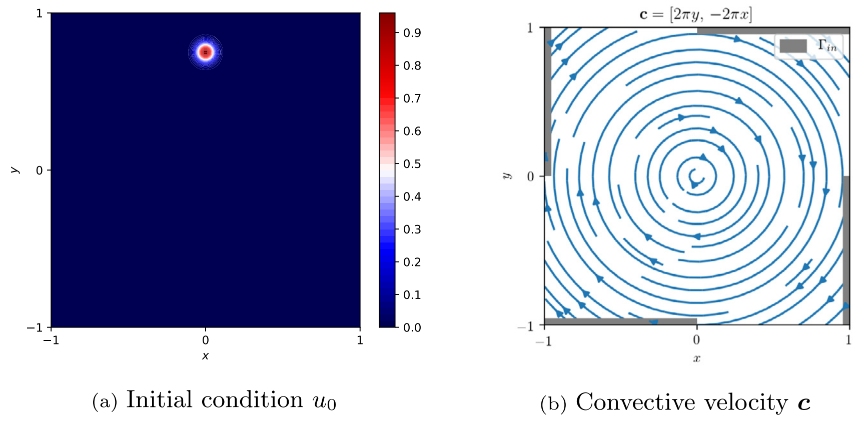

Fig. 18. Unsteady 2D Gaussian Pulse advection: Training test case setup. The locations of inflow boundary conditions are highlighted in gray.

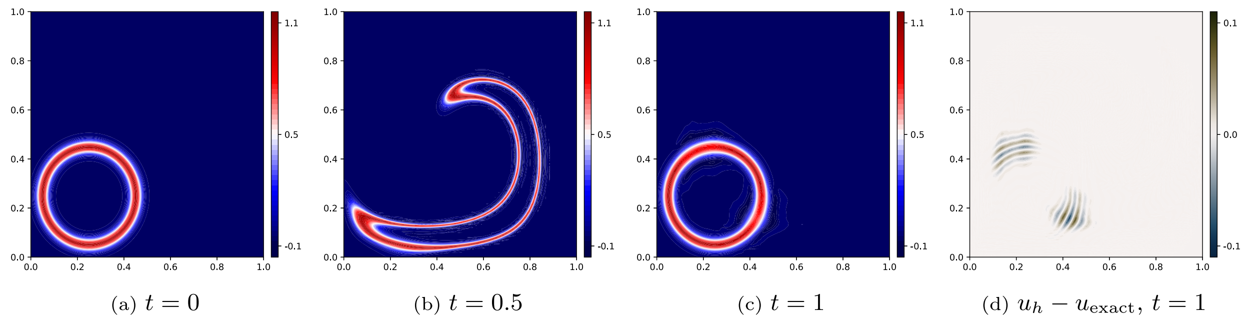

Fig. 19. Unsteady reversible 2D ring advection. Numerical solution on a 64×64 uniform grid.

where $\mu_x = 0$, $\mu_y = 0.75$, and $\sigma = 1/25$. The convective velocity is $\boldsymbol{c} = (2\pi y, -2\pi x)$ over the region $\Omega = (-1, 1)^2$, with numerical fluxes and upwind boundary conditions as specified in §3.2, specifically inflow boundary conditions $u_h = 0$ on Γ_{in} shown in (Fig. 18b) and no boundary conditions imposed on the outflow boundaries Γ_{out} (§3.2). We train the DRL agent for $3 \cdot 10^5$ training time steps on a budget of 200 cells; all other training parameters assume the default values (§3.5 and Appendix C).

Unsteady reversible 2D ring advection: deployment. Once trained, we deploy and benchmark the DRL agent's performance against a more complex unsteady linear advection (§3.2). Inspired by the novel advection-diffusion test case in [60], we consider the advection of a thin ring by a reversible swirl flow (Fig. 19). The initial tracer condition is given by

$$u_0 = \exp\left(-\frac{1}{2\sigma^2}\left(\sqrt{(x-x_0)^2 + (y-y_0)^2} - r_0\right)^2\right), \tag{16}$$

specifying a ring with inner radius r_0 and approximate thickness 3σ centered at the point (x_0, y_0). For this test case, we choose the parameters $r_0 = 0.2$, $(x_0, y_0) = (0.25, 0.25)$, and $3\sigma = 0.05$, describing a thin ring centered in the bottom right-hand corner of the domain, depicted in Fig. 19a. The velocity field is given by the reversible swirl flow

$$\boldsymbol{c}(x, y, t) = \left(\frac{3}{2}a(t)\sin^2(\pi x)\sin(2\pi y),\ -\frac{3}{2}a(t)\sin^2(\pi y)\sin(2\pi x)\right) \tag{17}$$

over the square domain $\Omega = (0, 1)^2$. Here $0 \le t \le 1$, $a(t) = 1$ if $t < 0.5$, and $a(t) = -1$ if $t \ge 0.5$, which reverses the direction of the flow field at $t = 0.5$. At the time $T = 1$, the analytical initial tracer distribution returns to its initial location due to the flow symmetry and can be compared to the initial condition to measure the error in the numerical solution. The inflow boundary conditions and forcing function are taken to be zero over the duration of the simulation; however, the locations of the inflow and outflow boundaries switch over the course of the time integration. The presence of steep gradients in the solution along the inner and outer circumferences of the ring during both the forward and backward swirl advection process makes this problem challenging, as well as a suitable test case to assess the ability of a numerical scheme to avoid spurious diffusion and dispersion, and to preserve features of the analytical solution [60].

All numerical simulations are run at polynomial order $p_{\text{order}} = 3$ and use a small time step $\Delta t = 10^{-3}$ over the LSERK45 time-marching so that the total error is dominated by the spatial discretization error.

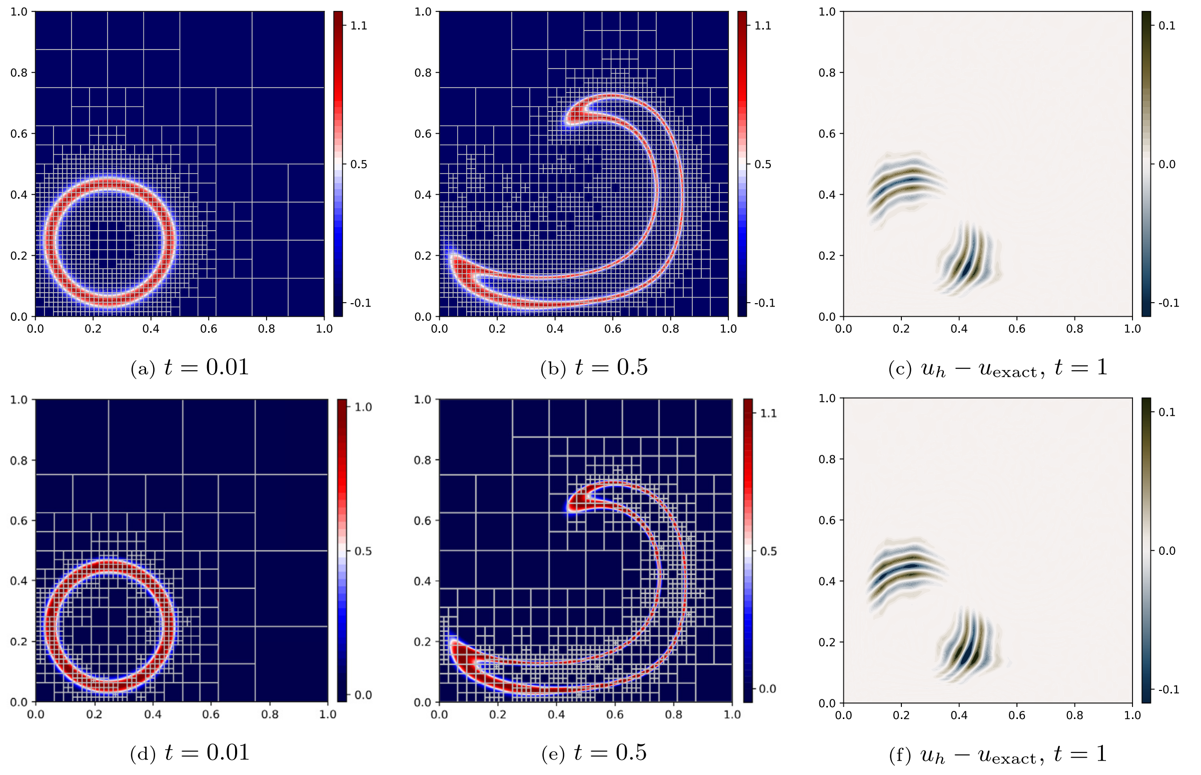

Fig. 20. Unsteady reversible 2D ring advection. AMR heuristic (top row) vs DRL agent (bottom row).

During deployment, we compare the trained DRL-AMR model to an AMR heuristic which uses the Kelly error indicator with a conservative `bulk`(60, 40) refinement strategy. Both strategies start on a fine, uniform mesh of 64×64 elements and are allowed 6 refinement cycles during initialization, after which, they are allowed one refinement cycle per time step for the duration of the simulation. In order to prevent the number of degrees of freedom from growing unbounded over time integration for the AMR heuristic, we apply an effective resolution requirement by specifying a maximum refinement depth of 4 as discussed in §3.4, corresponding to the finest possible grid of 64×64 cells. To provide a fair comparison, the RL policy is deployed with an equal maximum cell budget of $64^2 = 4096$ elements.

For baseline comparison for the AMR heuristic and the RL policy, we show first in Fig. 19 the numerical solution for a uniform grid of 64×64 elements, corresponding to the resolution limit for the AMR heuristic. Qualitatively, the uniform grid simulation is able to preserve the features of the ring over the forward and reverse advection, up to final integration time $t = 1$ (Fig. 19c).

The proposed meshes and resultant numerical solutions for the AMR heuristic and RL policy are shown in Fig. 20. As can be seen in panels 20a, 20b, 20d, and 20e, the qualitative features of the numerical solution resulting from both schemes are very similar to that of the uniform grid simulation (Fig. 19); that is, both approaches are able to successfully resolve the features of the solution. Similar to the other experiments, the RL agent is able to achieve comparable accuracy to the AMR heuristic, but with a comparatively coarser mesh, as is corroborated in Figs. 20c and 20f. The numerical errors for each approach are comparable and within 10 percent of the numerical error resulting from the uniform mesh. However, in this particular test case, at each time step, the RL-agent used far fewer elements.

The L^2-norm of the errors for the three numerical solutions at final time $T = 1$ are provided in Table 3 and the time series of active cells for both the AMR heuristic and the RL agent are shown in Fig. 21. They confirm that the DRL-AMR agent can achieve the same accuracy for a much smaller cell budget.

Table 3
Unsteady reversible 2D ring advection. Comparison of numerical errors at final time $T = 1$. The evolution in time of the number of active cells is provided in Fig. 21.

Refinement method	L^2-error	Percent change
Uniformly refined mesh (4096 cells), ground truth	$1.8316 \cdot 10^{-2}$	-
AMR Heuristic (Kelly)	$1.9553 \cdot 10^{-2}$	6.8%
DRL Agent	$1.9630 \cdot 10^{-2}$	7.2%

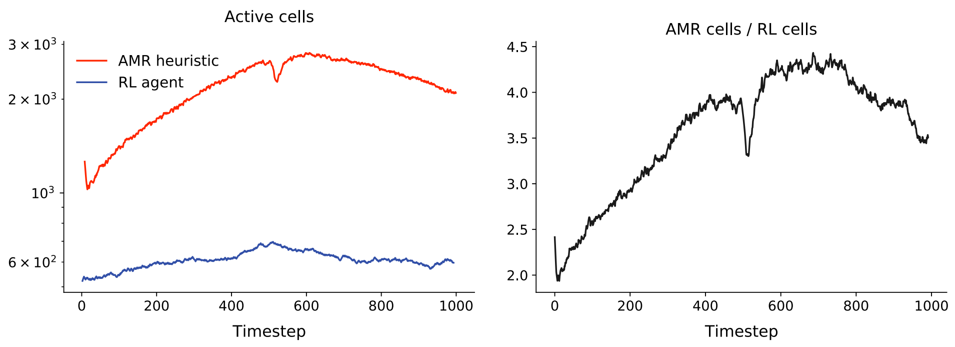

Fig. 21. Unsteady reversible 2D ring advection. Number of active cells versus time during deployment (post-initialization).

4.8. *Discussion of numerical experiments*

The numerical experiments in §4.1-§4.2 demonstrated a proof of concept for the DRL-AMR approach; we were able to train a model on a small budget and apply it effectively to larger, different problems governed by the same PDE. Our experiments indicated that random initialization led to a better exploration of the decision space and that all learning algorithms considered were able to solve the problem. We saw that for a simplified observation space, the neural network representing the trained RL policy was interpretable in terms of local solution conformity. The numerical experiments in §4.3-§4.4 showed that the method generalizes to time-dependent problems, as well as to different PDEs with mixed boundary conditions and significantly more complicated numerical schemes.

The test case in §4.5 demonstrated that the RL agent was able to learn a spatially heterogeneous, non-trivial strategy to increase accuracy per cost by preferentially refining around a problem region on the boundary, a strategy on which the AMR heuristic was not able to capitalize. The advection-diffusion problem in §4.6 extended the findings in §4.4 to higher dimensions and exhibited the ability of the RL policy to take advantage of the fast convergence of a post-processed solution as well as to resolve non-trivial features of a numerical solution cheaply. Lastly, the unsteady 2D advection problem in §4.7 displayed the ability of the trained RL policy to preserve similar non-trivial features over the course of time integration in problems with dynamic, sharp gradients.

Overall, our DRL-AMR methodology is flexible and effective at delivering efficient, high-quality solutions for both static and time-dependent problems over a wide range of different PDEs, boundary conditions, dimensions, and problem sizes.

5. Conclusions and future work

We introduced a novel deep reinforcement learning formulation for adaptive mesh refinement based on a partially observable Markov decision process designed to balance numerical accuracy with computational cost, with the goal of providing a learned, high-quality strategy for resolving solution features efficiently. The underlying idea is that rather than hand-designing a heuristic error indicator *a priori*, the reinforcement learning agent will instead learn one through trial and error during training by solving many inexpensive problems. This is advantageous because the learned policy network can be arbitrarily complex and make use of any feature included in the observation space, including information about computational cost. Furthermore, training the RL policy requires no domain-specific knowledge, as the relevant information is encoded in the numerical solver, and, more abstractly, in the underlying PDE. Conversely, specific features of a numerical scheme such as polynomial degree, convergence order, and problem dimension are implicitly considered in training through use of the numerical solver, as opposed to having to be analyzed and explicitly specified in the case of a manually-defined error indicator.

Our implementation shows that the resultant trained policies are able to execute adaptive refinement strategies which are competitive with, and in many cases, better than common AMR heuristics in terms of accuracy of the final solution per problem degree of freedom. Our methodology is not specific to any particular PDE, spatial dimension, or numerical scheme, and can flexibly incorporate physical or temporal history data into the observation space. The local nature of the RL problems allows for training the RL agent on small problems and deploying the policy on much larger problems, ensuring scalability. Lastly, at no point during training nor model deployment do we ever make use of an exact solution or a “ground truth.”

Future work could incorporate p-refinement into the learning process to allow the RL agent access to a richer set of finite element representations of the numerical solution, incorporate a more sophisticated belief distribution as to the regions of under- or over-resolution in the numerical solution, or integrate transfer learning using existing error estimators to accelerate training. Finally, the application of the new DRL-AMR schemes to vector-valued problems such as the Navier–Stokes equations and geophysical equations such as those used in storm surge [61,62], ocean [63–67], and atmospheric [68] applications, as well as studies of related numerical topics such as slope-limiting, preconditioning, and adaptive time integration, are the subject of ongoing research.

CRediT authorship contribution statement

C.F. conceived the idea, implemented the PDE solvers and RL methods, conducted the numerical experiments, and wrote the first draft of the manuscript, along with many of the subsequent revisions. A.C. conceived the tunable policies idea, provided valuable feedback related to the RL-formulation, and helped write several drafts of the manuscript. P.F.J.L. supervised the work, conceived of the ring advection test case, edited each iteration and revision of the manuscript, and provided guidance on the scope and structure of the manuscript and project as a whole.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

We thank the members of the MSEAS group, especially Dr. C. Mirabito for review of the manuscript. We are grateful to the Office of Naval Research (ONR) for partial research support under grants N00014-18-1-2781 (DRI-CALYPSO) and N00014-20-1-2023 (MURI ML-SCOPE) to the Massachusetts Institute of Technology. We also thank the Spring 2022 course staff of 2.C51 for helpful discussions related to machine learning for physical systems. C. Foucart would additionally like to thank the active `deal.II` software development community for helpful discussion related to cross-language software development.

Appendix A. Tunable policies

Computational resources are user- and time-dependent. For different applications, or even in the middle of a simulation, we may seek to adjust how aggressive our reinforcement learning agent is in refining or coarsening our mesh. We directly encode the trade-off between accuracy and computational cost in hyperparameter γ_c, which the user may tune freely. However, naively changing γ_c would require re-training our reinforcement learning agent. Next, we show how we can avoid re-training while simultaneously allowing for policy tuning.

The more general setting to this problem is multi-objective reinforcement learning (MORL). We seek an agent that is optimized over a continuum of objective functions depending upon γ_c. The value function $V_t^\pi(s_t)$ is the expected reward following policy π given state s_t. Traditionally, $V_t^\pi : S \to \mathbb{R}$ maps an observation to a scalar, but in MORL, the value function $V_t^\pi : S \to \mathbb{R}^d$ maps an observation to a d-dimensional vector indexed by the different objective functions. In our case, the value function $V_t^\pi(s_t; \gamma_c) : S \to \mathcal{C}_b(\mathbb{R})$ maps an observation to a bounded continuous function. Most approaches in the MORL literature attempt to find either a single policy by scalarizing the vector-valued V_t^π or multiple policies by repeatedly training over different objective functions (see [69] for a literature review). However, due to a particular feature of our expected reward function Q_t, we are able to simply learn over a continuum of objective functions.

Consider an expected reward function $Q_t : \mathcal{O} \times \mathcal{A} \to \mathbb{R}$ that can be split into two parts: a function that can be explicitly computed given the observation, i.e. the "known" function $Q^{(k)}$, and a function that needs to be learned, $Q^{(l)}$.

$$Q_t(s, a; \gamma_c) = Q_t^{(k)}(s, a; \gamma_c) + Q_t^{(l)}(s, a) \tag{18}$$

There is no reason to learn $Q^{(k)}$ if it can be explicitly computed; instead, we should just represent $Q^{(l)}$ with a deep neural network as in deep Q-learning. Then, the outputs of the two Q functions can be combined at the end, and the argmax of the resulting sum will be the action that we take. Importantly, the hyperparameter γ_c is only an argument of the known function that we can explicitly compute. This allows us to learn one function Q^l but then change our policy adaptively as a function γ_c. Computationally, we really learn the function $\hat{Q}_t : S \to \mathbb{R}^{|\mathcal{A}|}$ that maps an observation to the expected reward for every action in the action space, and we denote this modified and approximate expected reward function with a hat. In Fig. 22, we delineate a traditional deep Q-learning policy π from a tunable policy $\tilde{\pi}$.

In our particular case, the reward function (2) is naturally decomposed into a part that needs to be learned and one that can be explicitly computed: the accuracy needs to be learned but the computational cost and barrier function are explicitly known. Call these partial rewards $R^{(l)}$ and $R^{(k)}$, respectively. We can write our Q function as follows.

$$Q_t(s, a) = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma_k^i R_{t+i+1}^{(k)} + \sum_{i=0}^{\infty} \gamma^i R_{t+i+1}^{(l)} \middle| S_t = s, A_t = a \right] \tag{19}$$

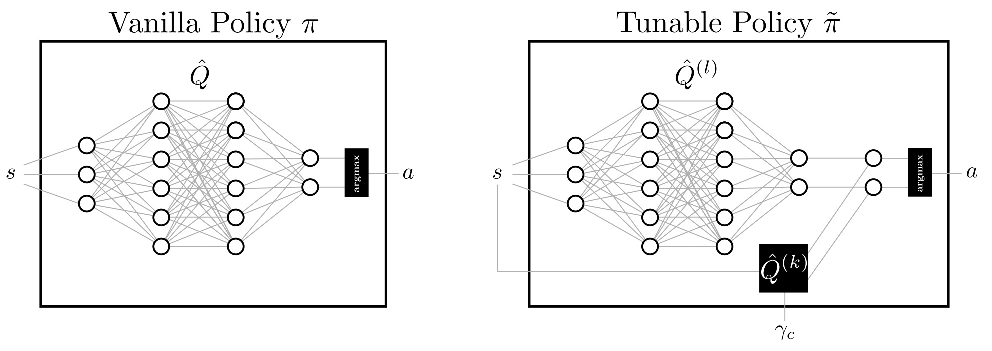

Fig. 22. We depict a vanilla policy based on Deep Q-Learning and a tunable policy. The tunable policy splits the $\hat{Q}$ function into known and learned functions. The learned function is represented by a neural network, while the known policy can be evaluated explicitly given the observations. In the tunable policy, the hyperparameter γ_c is an input to the known $\hat{Q}^{(k)}$ function which does not have to be re-learned.

Above, we have used different discount factors γ_k and γ for $R^{(k)}$ and $R^{(l)}$, respectively. We consider the case where $\gamma_k = 0$. In our example, setting the discount factor on the computational cost to zero is justified in that we are usually only concerned with the computational cost at a given instant. Furthermore, the computational load on a machine is often highly unpredictable; for example, other users may submit jobs in the middle of our simulation. As such, a greedy approach to discounting the computational penalty is desirable, and we may rewrite our expected reward as follows.

$$Q_t(s,a) = \underbrace{\mathbb{E}\left[R^{(k)}_{t+1} \middle| S_t = s, A_t = a \right]}_{Q^{(k)}_t} + \underbrace{\mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R^{(l)}_{t+i+1} \middle| S_t = s, A_t = a \right]}_{Q^{(l)}_t} \tag{20}$$

With that, we have defined our known and learned Q functions, and to train we simply omit $R^{(k)}$ from our reward entirely by setting $\gamma_c = 0$. Once trained, we append $Q^{(k)}$ to the end of our $Q^{(k)}$ network in order to make predictions, and γ_c may be set arbitrarily without retraining. We note that separable rewards are not limited to this case where we balance computational cost and simulation accuracy, and we posit that there are many other areas where this framework may be beneficial.

While we have described how to perform MORL in instances where Q is separable and the action space is discrete, this framework should be extendable to the continuous case. In actor-critic policies where the critic learns the Q function (e.g. deep deterministic policy gradient (DDPG) [70], twin delayed DDPG (TD3) [71], and soft actor critic (SAC) [72] methods), one should be able to apply a similar methodology: after learning $Q^{(l)}$, one should be able to append $Q^{(k)}$. In the continuous case, however, the actor must be re-trained since we cannot just take the argmax of a continuous action space. Fortunately, re-training the actor should be computationally inexpensive since the policy networks tend to be small, and we can just randomly sample from the action space. That is, we do not need to re-sample from our environment; instead, as we already know our Q function, we simply need to re-train the actor to maximize the estimated reward given an action from our action space. This procedure is the subject of ongoing research.

Appendix B. Exact solution, §4.6

The exact solution is given by

$$u(\boldsymbol{x}) = \sum_{i=1}^{25} \frac{1}{2\pi\sigma^2} \exp\left(-\frac{1}{\sigma^2} \|\boldsymbol{x} - \boldsymbol{x}_i\|^2 \right)$$

with parameter $\sigma = 1/10$ and source centers $\boldsymbol{x}_i$ described in Table 4.

Table 4
Description of source centers, where $\alpha = 1.1$.

i	$\boldsymbol{x_i} = (x_1, x_2/\alpha)$								
1	(-0.75, -0.25)	6	(-0.25, -0.25)	11	(0.125, -0.75)	16	(-0.75, -0.625)	21	(-0.25, -0.375)
2	(-0.75, -0.75)	7	(-0.25, -0.50)	12	(0.5, -0.75)	17	(-0.50, -0.625)	22	(0.125, -0.625)
3	(-0.75, -0.50)	8	(-0.25, -0.75)	13	(0.5, -0.5)	18	(-0.25, -0.625)	23	(0.5 , -0.625)
4	(-0.50, -0.25)	9	(0.125, -0.25)	14	(0.5 , -0.25)	19	(-0.75, -0.375)	24	(0.625 ,-0.25)
5	(-0.50, -0.50)	10	(0.125, -0.50)	15	(0.75 , -0.25)	20	(-0.50, -0.375)	25	(0.125, -0.375)

Appendix C. Additional training parameters

For all numerical experiments, we used a time discount factor $\gamma = 0.99$, which is the default for the Stable-Baselines-3 library. The large negative reward accrued by the agent upon running out of computational resources was taken to be $R_{\text{exceed}} = -1 \cdot 10^3$; this was important as a learning signal in the cases where the RL agent was trained on a small computational budget, as the barrier function $B(p)$ in (2) is undefined outside $p = 1$. Training episodes were terminated after a finite number of iterations, we used 200; this choice is arbitrary, as long as the number of iterations is large enough that it is possible for the agent to approach its computational budget and receive the large negative reward for doing so. The parameters used for RL training were given in Table 2.

References

[1] Susanne C. Brenner, L. Ridgway Scott, The Mathematical Theory of Finite Element Methods, vol. 3, Springer, 2008.
[2] Jan S. Hesthaven, Tim Warburton, Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications, Springer Science & Business Media, 2007.
[3] Christopher M. Bishop, Nasser M. Nasrabadi, Pattern Recognition and Machine Learning, vol. 4, Springer, 2006.
[4] Eurika Kaiser, J. Nathan Kutz, Steven L. Brunton, Sparse identification of nonlinear dynamics for model predictive control in the low-data limit, Proc. R. Soc. A 474 (2219) (2018) 20180335.
[5] Chinmay S. Kulkarni, Abhinav Gupta, Pierre F.J. Lermusiaux, Sparse regression and adaptive feature generation for the discovery of dynamical systems, in: F. Darema, E. Blasch, S. Ravela, A. Aved (Eds.), Dynamic Data Driven Application Systems, DDDAS 2020, in: Lecture Notes in Computer Science, vol. 12312, Springer, Cham, November 2020, pp. 208–216.
[6] Aditi Krishnapriyan, Amir Gholami, Shandian Zhe, Robert Kirby, Michael W. Mahoney, Characterizing possible failure modes in physics-informed neural networks, Adv. Neural Inf. Process. Syst. 34 (2021).
[7] Tomasz Plewa, Timur Linde, V. Gregory Weirs, et al., Adaptive Mesh Refinement-Theory and Applications, Springer, 2005.
[8] Dietrich Braess, Carsten Carstensen, Ronald H.W. Hoppe, Convergence analysis of a conforming adaptive finite element method for an obstacle problem, Numer. Math. 107 (3) (2007) 455–471.
[9] Patrik Daniel, Alexandre Ern, Iain Smears, Martin Vohralík, An adaptive hp-refinement strategy with computable guaranteed bound on the error reduction factor, Comput. Math. Appl. 76 (5) (2018) 967–983.
[10] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, Martin Kronbichler, Algorithms and data structures for massively parallel generic adaptive finite element codes, ACM Trans. Math. Softw. (TOMS) 38 (2) (2012) 1–28.
[11] Peter Binev, Wolfgang Dahmen, Ron DeVore, Adaptive finite element methods with convergence rates, Numer. Math. 97 (2) (2004) 219–268.
[12] Hans De Sterck, T. Manteuffel, S. McCormick, Joshua Nolting, John Ruge, Lei Tang, Efficiency-based h- and hp-refinement strategies for finite element methods, Numer. Linear Algebra Appl. 15 (2–3) (2008) 89–114.
[13] Ronald H.W. Hoppe, Yuri Iliash, Chakradhar Iyyunni, Nasser Hassan Sweilam, A posteriori error estimates for adaptive finite element discretizations of boundary control problems, 2006.
[14] Ronald H.W. Hoppe, Guido Kanschat, Tim Warburton, Convergence analysis of an adaptive interior penalty discontinuous Galerkin method, SIAM J. Numer. Anal. 47 (1) (2009) 534–550.
[15] Pedro Morin, Ricardo H. Nochetto, Kunibert G. Siebert, Data oscillation and convergence of adaptive fem, SIAM J. Numer. Anal. 38 (2) (2000) 466–488.
[16] Richard I. Klein, Star formation with 3-d adaptive mesh refinement: the collapse and fragmentation of molecular clouds, J. Comput. Appl. Math. 109 (1–2) (1999) 123–152.
[17] Mark Ainsworth, J. Tinsley Oden, A posteriori error estimation in finite element analysis, Comput. Methods Appl. Mech. Eng. 142 (1–2) (1997) 1–88.
[18] Ivo Babuska, John Whiteman, Theofanis Strouboulis, Finite Elements: an Introduction to the Method and Error Estimation, Oxford University Press, 2010.
[19] Alexandre Ern, Annette F. Stephansen, Martin Vohralík, Guaranteed and robust discontinuous Galerkin a posteriori error estimates for convection–diffusion–reaction problems, J. Comput. Appl. Math. 234 (1) (2010) 114–130.
[20] Donald W. Kelly, J.P. De S.R. Gago, Olgierd C. Zienkiewicz, I. Babuska, A posteriori error analysis and adaptive processes in the finite element method: part I—error analysis, Int. J. Numer. Methods Eng. 19 (11) (1983) 1593–1619.
[21] Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Marc Fehling, Rene Gassmöller, Timo Heister, Luca Heltai, Uwe Köcher, Martin Kronbichler, Matthias Maier, et al., The deal. II library, version 9.3, J. Numer. Math. 29 (3) (2021) 171–186.
[22] Jesus Bonilla, Santiago Badia, Monotonicity-preserving finite element schemes with adaptive mesh refinement for hyperbolic problems, J. Comput. Phys. 416 (2020) 109522.
[23] Amit Joshi, Alan B. Thompson, Eva M. Sevick-Muraca, Wolfgang Bangerth, Adaptive finite element methods for forward modeling in fluorescence enhanced frequency domain optical tomography, in: Biomedical Topical Meeting, Optical Society of America, 2004, page WB7.
[24] Antoni Vidal-Ferràndiz, Amanda Carreño, Damián Ginestar Peiro, Gumersindo Jesús Verdú Martín, hp-finite element method for the simplified PN equations, in: Congreso de Métodos Numéricos en Ingeniería (CMN 2017), Actas, International Center for Numerical Methods in Engineering (CIMNE), 2017, pp. 208–220.
[25] Tony D. Young, Rickard Armiento, Strategies for h-adaptive refinement for a finite element treatment of harmonic oscillator Schrödinger eigenproblem, Commun. Theor. Phys. 53 (6) (2010) 1017.
[26] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, Anil Anthony Bharath, Deep reinforcement learning: a brief survey, IEEE Signal Process. Mag. 34 (6) (2017) 26–38.
[27] Richard S. Sutton, Andrew G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[28] Kurt Hornik, Maxwell Stinchcombe, Halbert White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (5) (1989) 359–366.
[29] Sho Sonoda, Noboru Murata, Neural network with unbounded activation functions is universal approximator, Appl. Comput. Harmon. Anal. 43 (2) (2017) 233–268.
[30] Martin Kronbichler, The discontinuous Galerkin method: derivation and properties, in: Martin Kronbichler, Per-Olof Persson (Eds.), Efficient High-Order Discretizations for Computational Fluid Dynamics, Springer International Publishing, 2021, pp. 1–55.
[31] Bernardo Cockburn, Discontinuous Galerkin methods, ZAMM-J. Appl. Math. Mech. (Zeitschrift für Angewandte Mathematik und Mechanik: Applied Mathematics and Mechanics) 83 (11) (2003) 731–754.
[32] G. Gassner, C. Altmann, F. Hindenlang, M. Staudenmeier, C.D. Munz, Explicit discontinuous Galerkin schemes with adaptation in space and time, in: 36th CFD/ADIGMA Course on hp-Adaptive and hp-Multigrid Methods, VKI LS, 2009.
[33] Lilia Krivodonova, Joseph E. Flaherty, Error estimation for discontinuous Galerkin solutions of two-dimensional hyperbolic problems, Adv. Comput. Math. 19 (1) (2003) 57–71.

[34] Lilia Krivodonova, Jianguo Xin, J-F. Remacle, Nicolas Chevaugeon, Joseph E. Flaherty, Shock detection and limiting with discontinuous Galerkin methods for hyperbolic conservation laws, Appl. Numer. Math. 48 (3–4) (2004) 323–338.
[35] Fabio Naddei, Marta de la Llave Plata, Vincent Couaillier, Frédéric Coquel, A comparison of refinement indicators for p-adaptive simulations of steady and unsteady flows using discontinuous Galerkin methods, J. Comput. Phys. 376 (2019) 508–533.
[36] Jan Bohn, Michael Feischl, Recurrent neural networks as optimal mesh refinement strategies, Comput. Math. Appl. 97 (2021) 61–76.
[37] Amir-massoud Farahmand, Saleh Nabi, Daniel N. Nikovski, Deep reinforcement learning for partial differential equation control, in: 2017 American Control Conference (ACC), IEEE, 2017, pp. 3120–3127.
[38] Yufei Wang, Ziju Shen, Zichao Long, Bin Dong, Learning to discretize: solving 1D scalar conservation laws via deep reinforcement learning, arXiv preprint, arXiv:1905.11079, 2019.
[39] Shiyin Wei, Xiaowei Jin, Hui Li, General solutions for nonlinear differential equations: a rule-based self-learning approach using deep reinforcement learning, Comput. Mech. 64 (5) (2019) 1361–1374.
[40] Jie Pan, Jingwei Huang, Gengdong Cheng, Yong Zeng, Reinforcement learning for automatic quadrilateral mesh generation: a soft actor-critic approach, arXiv preprint, arXiv:2203.11203, 2022.
[41] Jiachen Yang, Tarik Dzanic, Brenden Petersen, Jun Kudo, Ketan Mittal, Vladimir Tomov, Jean-Sylvain Camier, Tuo Zhao, Hongyuan Zha, Tzanio Kolev, et al., Reinforcement learning for adaptive mesh refinement, arXiv preprint, arXiv:2103.01342, 2021.
[42] Niklas Fehn, Wolfgang A. Wall, Martin Kronbichler, On the stability of projection methods for the incompressible Navier–Stokes equations based on high-order discontinuous Galerkin discretizations, J. Comput. Phys. 351 (2017) 392–421.
[43] M.P. Ueckermann, P.F.J. Lermusiaux, High order schemes for 2D unsteady biogeochemical ocean models, Ocean Dyn. 60 (6) (December 2010) 1415–1445, https://doi.org/10.1007/s10236-010-0351-x.
[44] Niklas Fehn, Wolfgang A. Wall, Martin Kronbichler, Robust and efficient discontinuous Galerkin methods for under-resolved turbulent incompressible flows, J. Comput. Phys. 372 (2018) 667–693.
[45] Yuxi Li, Deep reinforcement learning: an overview, arXiv preprint, arXiv:1701.07274, 2017.
[46] Karl Johan Åström, Optimal control of Markov processes with incomplete state information, J. Math. Anal. Appl. 10 (1) (1965) 174–205.
[47] David Silver, Lectures on reinforcement learning, https://www.davidsilver.uk/teaching/, 2015.
[48] Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, David Wells, The deal. II finite element library: design, features, and insights, Comput. Math. Appl. 81 (2021) 407–422.
[49] Walter A. Strauss, Partial Differential Equations: An Introduction, John Wiley & Sons, 2007.
[50] Pierre F.J. Lermusiaux, Numerical fluid mechanics, MIT OpenCourseWare, https://ocw.mit.edu/courses/mechanical-engineering/2-29-numerical-fluid-mechanics-spring-2015/lecture-notes-and-references/, May 2015.
[51] Andrei D. Polyanin, Vladimir E. Nazaikinskii, Handbook of Linear Partial Differential Equations for Engineers and Scientists, CRC Press, 2015.
[52] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, Playing Atari with deep reinforcement learning, https://arxiv.org/abs/1312.5602, 2013.
[53] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu, Asynchronous methods for deep reinforcement learning, https://doi.org/10.48550/ARXIV.1602.01783, 2016, https://arxiv.org/abs/1602.01783.
[54] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, Proximal policy optimization algorithms, https://arxiv.org/abs/1707.06347, 2017.
[55] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Noah Dormann, Stable Baselines3, 2019.
[56] Mark H. Carpenter, Christopher A. Kennedy, Fourth-order 2N-storage Runge-Kutta schemes, Technical report, NASA, 1994.
[57] Ngoc Cuong Nguyen, Jaume Peraire, Bernardo Cockburn, An implicit high-order hybridizable discontinuous Galerkin method for linear convection–diffusion equations, J. Comput. Phys. 228 (9) (2009) 3232–3254.
[58] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba, OpenAI Gym, arXiv preprint, arXiv:1606.01540, 2016.
[59] Bernardo Cockburn, Bo Dong, Johnny Guzmán, Optimal convergence of the original DG method for the transport-reaction equation on special meshes, SIAM J. Numer. Anal. 46 (3) (2008) 1250–1265.
[60] Chinmay S. Kulkarni, Pierre F.J. Lermusiaux, Advection without compounding errors through flow map composition, J. Comput. Phys. 398 (December 2019) 108859, https://doi.org/10.1016/j.jcp.2019.108859.
[61] Kyle T. Mandli, Clint N. Dawson, Adaptive mesh refinement for storm surge, Ocean Model. 75 (2014) 36–50.
[62] Ajimon Thomas, J.C. Dietrich, M. Loveland, A. Samii, C.N. Dawson, Improving coastal flooding predictions by switching meshes during a simulation, Ocean Model. 164 (2021) 101820.
[63] Patrick J. Haley Jr., Pierre F.J. Lermusiaux, Multiscale two-way embedding schemes for free-surface primitive equations in the "Multidisciplinary Simulation, Estimation and Assimilation System", Ocean Dyn. 60 (6) (December 2010) 1497–1537, https://doi.org/10.1007/s10236-010-0349-4.
[64] P.J. Haley Jr., A. Agarwal, P.F.J. Lermusiaux, Optimizing velocities and transports for complex coastal regions and archipelagos, Ocean Model. 89 (2015) 1–28, https://doi.org/10.1016/j.ocemod.2015.02.005.
[65] M.P. Ueckermann, P.F.J. Lermusiaux, Hybridizable discontinuous Galerkin projection methods for Navier–Stokes and Boussinesq equations, J. Comput. Phys. 306 (2016) 390–421, https://doi.org/10.1016/j.jcp.2015.11.028.
[66] C. Foucart, C. Mirabito, P.J. Haley Jr., P.F.J. Lermusiaux, Distributed implementation and verification of hybridizable discontinuous Galerkin methods for nonhydrostatic ocean processes, in: OCEANS Conference 2018, Charleston, SC, IEEE, October 2018.
[67] Corbin Foucart, Chris Mirabito, Patrick J. Haley Jr., Pierre F.J. Lermusiaux, High-order discontinuous Galerkin methods for nonhydrostatic ocean processes with a free surface, in: OCEANS 2021 IEEE/MTS, IEEE, September 2021, pp. 1–9.
[68] Michal A. Kopera, Francis X. Giraldo, Analysis of adaptive mesh refinement for IMEX discontinuous Galerkin solutions of the compressible Euler equations with application to atmospheric simulations, J. Comput. Phys. 275 (2014) 92–117.
[69] Conor F. Hayes, Roxana Rădulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M. Zintgraf, Richard Dazeley, Fredrik Heintz, et al., A practical guide to multi-objective reinforcement learning and planning, Auton. Agents Multi-Agent Syst. 36 (1) (2022) 1–59.
[70] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra, Continuous control with deep reinforcement learning, https://arxiv.org/abs/1509.02971, 2015.
[71] Scott Fujimoto, Herke van Hoof, David Meger, Addressing function approximation error in actor-critic methods, in: Jennifer Dy, Andreas Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, 10–15 Jul 2018, in: Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 1587–1596, https://proceedings.mlr.press/v80/fujimoto18a.html.
[72] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine, Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor, CoRR, arXiv:1801.01290 [abs], 2018, http://arxiv.org/abs/1801.01290.