Extended Abstract

Motivation Solving elliptic partial differential equations (PDEs) efficiently is useful for various computational applications such as computer graphics, physics simulations, and geometric processing. Traditional numerical methods such as finite difference/finite element methods are commonly used to solve PDEs, but struggle with irregular geometries or high-dimensional problems. Recently, new developments in Monte Carlo-based methods offer new mesh-free approaches. However, these methods result in high variance at low iterations and require a high computational budget. To more effectively run Monte-Carlo solvers, we seek to utilize reinforcement learning as a potential approach to adaptively optimize Monte Carlo sampling strategies.

Method This project investigates the integration of reinforcement learning agents within Monte Carlo PDE solvers to improve sampling efficiency and accuracy, which has not been done before. We focus on the Poisson equation as an elliptic PDE. To solve these PDEs, we reformulate the PDE as an integral and then carry out Monte Carlo random walks to approximate the solutions to the integrals. To augment this process with RL, we first utilize an iterative strategy where an agent will take in a Monte Carlo solution, select a few points to carry out additional Monte Carlo random walks on, on which the solver then carries out these walks on. This is an iterative process for some n iterations dependent on the computational budget. An alternative, faster strategy that was also implemented consists of using an RL agent as a sampling importance network, where at each grid point, the agent picks some action that corresponds to a certain number of Monte Carlo walks. Then, these walks are carried out before being passed through a neural network denoiser to output the final solution.

Implementation We focus on two RL algorithms in this study: Deep Q-learning and Proximal Policy Optimization (PPO). For the iterative adaptive sampling strategy, we implement both Deep Q-learning and PPO. At each iterative step, the agent is taught to select some number of grid points where a fixed additional number of random walks will be carried on. This corresponds to taking the top K Q-values or top K probabilities from the agents. The reward was equal to the drop in MSE of the solution from one iteration to the next. For the second strategy utilizing a hybrid RL-Neural Network setup, we utilize PPO to train the RL agent to determine how many random walks are carried out at each block of grid points, where each block consists of a 16x16 square of grid points. After carrying out the random walks, the result was passed through a neural network denoiser that outputted a final solution. The neural network was trained using MSE as its loss function, and the reward given to the agent was $10^{1-\text{NN loss}}$.

Results For the adaptive strategy, the Deep Q-learning agent was able to generate a better solution compared to uniform sampling at low computational budgets (0.49 vs. 0.84 MSE at 1000 total walks). However, at a larger number of walks (1e7), Deep Q-Learning, PPO, and uniform sampling began to converge. For our hybrid RL-neural network setup, our agent was able to achieve an MSE better than that from a 1000th iteration WoS solution, while only taking an average of 90 iterations as input, leading to a >10x decrease in computational budget.

Discussion Our results highlight the potential of reinforcement learning as a strategy to make Monte Carlo PDE solvers more efficient through adaptive sampling. We see that our hybrid approach leads to substantial efficiency improvements over a baseline Monte Carlo WoS solver. However, challenges and limitations still remain in training a more general reinforcement learning agent that can extend beyond Poisson's equation and the unit square, as fine-tuning hyperparameters to allow the agent to learn was especially difficult.

Conclusion We show that integrating RL techniques and neural network denoising in tandem into Monte-Carlo PDE solvers can significantly improve the computational efficiency of these solvers. This can allow for improved efficiency when solving PDEs for computer graphics applications while maintaining solution quality. Future work can extend our approaches to expand the agent action space, refining input features, and also exploring other architectural changes and rewards to improve robustness and adaptability.

Deep Reinforcement Learning for Efficient PDE Solvers

Ivan Ge
Department of Physics
Stanford University
`ivange@stanford.edu`

Abstract

Efficiently solving elliptic partial differential equations (PDEs) is essential for various applications in computer graphics, simulations, and geometric processing. Traditional methods like FEM or FDM often struggle with complex shapes and require meshes. Newer methods like Monte Carlo solvers offer mesh-free approaches that are more scalable; however, these random walk-based methods face slow convergence and high variance at low iterations. This paper proposes an initial study of utilizing reinforcement learning techniques to optimize the sampling strategies of Monte Carlo-based PDE solvers beyond uniform sampling. We explore both iterative RL-guided sampling strategies and also hybrid RL-Neural Network denoiser strategies. We find that our hybrid method is able to achieve similar high-quality solutions while using much less computation compared to baseline uniform Monte Carlo methods. These results demonstrate the applicability of reinforcement learning to Monte Carlo PDE solvers.

1 Introduction

Elliptic partial differential equations (PDEs) form the mathematical foundation for numerous computational processes in computer graphics, geometric processing, and scientific simulations [4, 8]. Solving these elliptic PDEs fast and accurately is useful for computer graphics applications ranging from 3D reconstruction to animation techniques. In particular, they are critical for surface smoothing, mesh generation, and image inpainting among other applications in physics such as fluid dynamics and heat transfer simulations [3, 9, 32]. The ability of solve elliptic PDEs efficiently and accurately directly impacts the quality and computational feasibility of a variety of graphics techniques.

Traditional numerical methods including finite different methods (FDM), finite element methods (FEM), or other grid-based solvers have been used to solve elliptic PDEs for decades [5, 6, 33]. However, these methods all require explicit mesh generation and spatiotemporal discretization, making them problematic in certain graphics applications where these conditions cannot be met [30, 31]. Furthermore, these methods struggle with irregular geometries and boundary conditions, while also exhibiting poor scalability for high-dimensional problems/domains [10, 12].

Recent advances in Monte Carlo methods for PDE solvers have introduced new techniques to address the limitations of traditional solvers [18, 23]. They performs fundamentally different computational approach that utilize probabilistic methods to generate solutions to various PDEs. Current Monte Carlo-based PDE solvers, such as Walk-on-Spheres [26], provide an unbiased and mesh-free method for graphics and simulation processes. These methods reformulate elliptic PDEs as recursive integrals that are solved using Monte Carlo sampling, and allow solutions to be obtained without explicit grid generation. Other methods like Walk-On-Stars have also been developed to allow these solvers to handle more complex boundary conditions and irregular domains [27].

Despite their advantages, Monte Carlo PDE solvers still face significant practical challenges. At low sample counts, these methods suffer extremely high variance and can result in noisy solutions unless an extremely large number of iterations are run. Especially for elliptic PDEs, random walk processes can become arbitrarily long and result in exponential variance growth [18]. To increase solver efficiency, previous methods have explored utilizing neural fields to terminate long random walks by sampling low-iteration solutions [15, 37]. Other methods have used neural networks to "predict" higher iteration solutions from low-iteration solutions [16], effectively using machine learning as a variance reduction technique (denoiser) [16, 21]. These methods have achieved modest speedups, but require extensive training data and are less adaptable to varying computational requirements and different problem domains.

The introduction of reinforcement learning (RL) techniques into numerical simulation is an emerging paradigm that offers the potential to address the limitations of current Monte Carlo PDE solvers [19, 34]. RL-based adaptive sampling algorithms have shown some success in sampling complex spaces by learning relative importance of different regions within the state space [11, 17]. Unlike supervised learning approaches that require extensive training datasets, RL can learn optimal sampling strategies through direct iteration with the PDE solving process [14]. This allows for the development of adaptive agents that can dynamically adjust their sampling behavior based on local solution characteristics, convergence rates, as well as overall computational budgets [1]. This sequential decision-making framework for RL aligns with the iterative nature of Monte Carlo PDE solving, making their combination a potential solution for solving elliptic PDEs.

This project aims to accelerate current Monte Carlo (MC) methods for solving elliptic partial differential equations (PDEs) by integrating reinforcement learning (RL) agents into the solving process.

2 Related Work

A variety of Monte Carlo algorithms have been developed to solve elliptic PDEs dating back to the mid-1900s. The classic Walk-on-Spheres (WoS) algorithm utilizes random walks on spheres to estimate solutions to elliptic PDEs [20]. Recent works have expanded to Walk-On-Stars to extend the initial WoS method to work for mixed Dirichlet and Neumann boundary conditions, also in a grid-free manner. These approaches are discretization-free and unbiased, which allows them to be run on various geometries, which is useful for graphics and simulation applications [24]. However, these methods have very slow convergence, meaning that a high order of magnitude number of samples are needed. The high variance of these solutions limit their practicality in real-time or high-resolution scenarios. Other traditional numerical solvers include finite element methods, which become expensive for meshes and high-dimensional PDEs [24].

In addition to Monte Carlo methods, researchers have also explored utilizing physics-informed neural networks (PINNs) and other deep learning approaches to train neural networks to generate solutions satisfying PDE equations and boundary conditions [2, 13]. These methods seek to train neural networks to satisfy both the PDE and boundary conditions through specialized loss functions [35]. However, in general, neural networks suffer from training difficulties such as highly non-convex loss landscapes, ill-conditioned gradients, and spectral bias [22, 36], while also tending produce biased solutions [16]. These challenges restrict the applicability of applying NNs across domains of different PDEs and boundary conditions.

Recent works have thus begun to integrate neural networks with Monte Carlo PDE solvers to achieve unbiased and low-noise solutions. Most recently, [16] utilized neural fields as a variance reduction tool inside a Monte Carlo solver. In their approach, lightweight neural networks were first trained to approximate the PDE solution over the domain, which was then used as a cache during Monte Carlo sampling. While they were able to achieve lower error than plain WoS solvers for limited computational budgets, these methods require manual hyperparameter tuning and are restricted to certain PDEs and geometries.

The project intends to introduce reinforcement learning into this context of hybrid Monte Carlo Neural Network PDE solvers. RL has shown promise for adaptive algorithms in scientific computing, with research applying RL-based adaptive samplers for Monte Carlo path tracing [28] as well as an adaptive Q-learning algorithm to solve elliptic PDEs [7]. This motivates the integration of RL techniques with hybrid Monte Carlo-neural network appraisees to achieve more robust and adaptive

PDE solvers. It suggests that RL has the potential to be leveraged to guide an MC PDE solver to achieve a high fidelity solution in few iterations for a broader number of PDEs and boundary conditions.

3 Baseline Method

In this paper, the elliptic PDEs that we are utilizing take the following form, as cited from [16]:

For $x \in \Omega$:

$$\nabla \cdot (\alpha(x)\nabla u(x)) + \omega(x)\nabla u(x) - \sigma(x)u(x) = -f(x)$$

For $x \in \partial\Omega$

$$u(x) = g(x)$$

where $\alpha, \omega,$ and σ are spatially varying coefficients, f is the forcing function, g is our boundary function, and Ω is defined by the signed distance function. To simplify our experiments and the generation of PDE solutions, we let $\omega = \sigma = 0$ and $\alpha = 1$ such that we recover Poisson's equation of

$$\nabla^2 u = -f(x)$$

To iteratively evaluate a solution, Monte Carlo PDE solvers express the numerical solution of a PDE using an integral. The solution to this type of PDE is as follows as

$$u(x) = S(x) + \int_{B_r(x)} u(y)G_x(y)dy + \int_{\partial B_r(x)} u(z)K_X(z)dz$$

where one integral is evaluated over a ball centered at x with radius r and the other is evaluated over the surface boundary of the ball. The other terms $S(x)$, $G_X(y)$, and $K_X(z)$ are dependent on Green's function and the Poisson kernel. More detailed definitions are provided in [16] and [24]. The Monte Carlo PDE solver method roughly works in the following steps:

1. For each grid point x (note that WoS methods don't explicitly need a grid, but we use a grid for ease of visualization and evaluation), construct the largest possible sphere $B_r(x)$ centered at x and remains within the boundary $\delta\Omega$.
2. Randomly sample a point y on the surface of the sphere $\partial B_r(x)$. If y is within ϵ of the boundary, then $u(y) \approx g(y)$. Otherwise, repeat the process from step 1, except treating y as the new point.
3. Once the walk terminates when it gets close enough to the boundary (note again that this walk could theoretically go on forever, so we often terminate after some large number of walks), the final boundary value $g(z)$ becomes one sample estimate of the original integral.

The main idea of such as method is that

$$u(x) = S(x) + \mathbb{E}[u(Y)]$$

where Y is a random variable representing the next point in the random walk, and its distribution follows Green's function and the Poisson kernel.

4 Experimental Setup

For the dataset, each individual PDE defined with a forcing function, boundary condition, and a reference solution. The reference solution is either an analytical solution (contructed for 100 PDEs), or a 10,000th iteration Monte Carlo Walk-on-Spheres (WoS) solution (constructed for 1000 PDEs with no analytical solution). Th first 100 PDEs are solved on a 32x32 grid and the 1000 PDEs are solved on a 128x128 grid, both on the unit square ($[0, 1] \times [0, 1]$). We carry out two sets of experiments as defined below.

1. For the first set of experiments, we utilize a custom-made dataset of 100 PDEs with analytical solutions u. For simplicity, these PDEs mainly consist of sinusoidal functions oscillating

across the domain ($A\sin(nx)\sin(ny)$). Since we have direct access to the boundary and forcing functions, we construct a Poisson WoS solver environment such that upon specification of the grid index and number of random walks, our solver can carry out the appropriate number of Monte Carlo samples and return the result. This experiment setup seeks to answer the following question:

Can an RL agent learn to iteratively choose certain solution grid points to carry out additional random walks on to improve solution quality?

To answer this question, we train two RL agents – a Deep Q-learning agent and a Proximal Policy Optimization (PPO) agent to carry this out. Both agents take in the current state, an estimate of the error $\Delta u - f$, and the number of random walks carried out at each grid point. Then, the agent outputs $n = \{10, 25, 50, 100\}$ new grid points to sample $m = \{10, 100\}$ new random walks on. It repeats this step for some number of episodes, dependent on the computational budget (defined as the total number of random walks carried out).

2. For the second set of experiments, we utilize the second dataset consisting of 1000 PDEs without analytical solution. This dataset is from [], although slight modifications have been made to remove particular PDEs that are not well-defined (i.e. always 0). A random 700/200/100 train/val/test split was generated. For each of the 1000 PDEs, the dataset has sampled MC iterations consisting of $1, 5, 10, 20, 50, 100, 1000, 10000$ iterations, where the 10000th iteration is used as the ground truth. From experiment 1, we noted that any pure sampling agent suffers from a lower-bound of the Monte Carlo WoS solver itself, namely the fact that the variance of the solution decreases $\propto \frac{1}{\sqrt{\text{num walks}}}$. Thus, our follow-up question becomes:

   Can an RL agent in tandem with a neural network denoiser learn per-pixel importance of the solution grid and sample appropriately?

   Here, we adopt an approach similar to [28] where our RL agent acts as a one-shot sampling importance network. For each grid point of the solution, the agent chooses how many random walks are carried out. Then, we carry out those number of walks at each point, and then the solution from that is fed into a UNet denoiser model, outputting the final solution.

We evaluate our methods based on two metrics that are common in computer graphics, namely the mean squared error (MSE) and peak signal-to-noise ratio (PSNR). The specific details of the models, including the hyperparameters, model architecture, and training regime are described in the Appendix.

5 Reinforcement Learning Methods

5.1 Experiment 1

For our first set of experiments, we utilize the following iterative setup. In brief, our input state into our agent consists of the current state, current visit counts, and residual, calculated as

$$r_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} - f_{i,j}$$

where h is the grid spacing in this case. In situations without a grid, perhaps other finite-element residual or meshless local polynomial stencils can be utilized based on neighboring points. Then, our agent outputs new grid points for an additional number of walks, which is then sampled. The whole process is then repeated. A diagram is shown in Figure 1.

We implement both PPO and Deep Q-learning, as PPO can provide stable updates to our agent as it is learning in a complex environment, and Deep Q-learning is optimal for the discrete actions that we are taking.

5.2 Experiment 2

For our second set of experiments, we utilize the setup shown in Fig. 4. This system implements a two-stage hybrid approach that combines an RL agent for adaptive sampling with a convolutional neural network to output the final PDE solution. The agent is given an input of a first iteration Monte

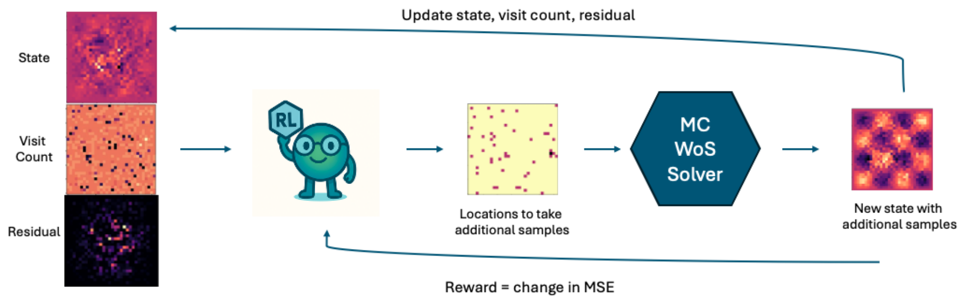

Update state, visit count, residual
State
Visit Count
Residual
RL
Locations to take additional samples
MC WoS Solver
New state with additional samples
Reward = change in MSE

Figure 1: Iterative RL-based sampling for Monte Carlo Walk-on-Spheres (WoS) solver

Carlo solution as well as a "variance" measure, denoted as the difference between the 1st and 5th iterations. The agent's action space is restricted to one action type (5, 10, 20, 50, 100 or 1000 steps) for each 16x16 block. This results in 64 independent decisions per image. The solver then carries out these walks, outputting a sampled solution, which is then fed through a neural network to output the final solution. The reward is calculated as $10^{1\text{ - loss}}$ as inspired from [28] where the loss is the final MSE loss from the neural network. We utilize PPO to provide stable updates as we are jointly training the neural network and RL agent. A diagram is shown in Figure 2.

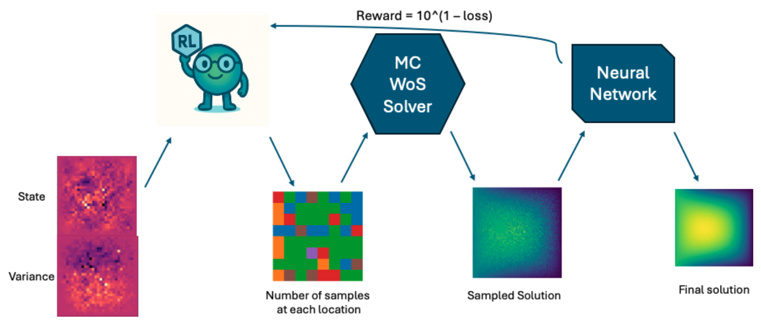


Figure 2: One-shot RL-based sampling for hybrid Monte Carlo-Neural Network solver

We trialed three different versions of this setup:

1. v1: Train RL agent and neural network together, and directly run inference.
2. v2: Train RL agent and neural network together, but run a few "fine-tuning" steps throughout and at the end by freezing the RL agent and training the neural network alone for an additional ~ 20 epochs.
3. v3: First, warm up neural network for ~ 10 epochs with the goal of providing the RL agent a more stable reward function. Then, train the RL agent and neural network.

6 Results

6.1 Baseline

For our Monte Carlo PDE solver baseline, we utilize a JAX-implementation of Walk of Spheres based on [16, 25]. Our domain is centered at $(0.5, 0.5)$ and defined on the unit square. For the dataset consisting of 1000 PDEs, we give an example of the WoS results after 1, 5, 10, and 20 iterations in Figure. To justify the designation of the 10000th iteration as the baseline, the WoS solver was ran on PDEs with analytical solutions for 10000 iterations, noting that the MSE of the solution was $< 10^{-5}$, meaning it suffices as a ground truth for our experiments. Two statistics are utilized to evaluate our results, namely the mean square error (MSE) and peak signal-to-noise ratio (PSNR). The metrics for our 1000 PDE dataset is shown below.

Metric	1	10	20	100	1000
MSE	0.063	0.020	0.016	0.0063	0.00069
PSNR	16.72	21.48	22.73	26.68	36.31

Table 1: MSE vs. Number of Iterations for Uniform Baseline

6.2 Experiment 1

The overall results from this agent trained with Deep Q-learning and PPO are shown below. Specific results from each method are further described in the respective subsections.

Total Num. Walks	1e4 (v1)	1e4 (v2)	1e4 (v3)	1e5	1e6	1e7
DQN MSE	**0.4901**	1.402	**0.6694**	**0.0719**	0.03511	**0.00129**
PPO MSE	1.405	2.362	0.8466	0.0809	0.04010	0.00142
Uniform MSE	0.8445	**0.8445**	0.8445	0.07345	**0.03341**	0.00137

Table 2: MSE vs. total Walk-on-Spheres samples. The DQN, PPO, and Uniform results are shown above. v1, v2, and v3 refer to different sampling methods that were used. For v1, 10 iterations were carried out, with each iteration sampling 100 spots with 10 random walks each. For v2, 20 iterations were carried out, with each iteration sampling 50 spots with 10 random walks each. For v3, 10 iterations were carried out, with each iteration sampling 50 spots with 20 random walks each.

6.2.1 Deep Q-Learning

The Deep Q-Learning agent results are shown in the first line of Table 2. We notice that initially, our DQN agent performs better than the uniform MSE for a smaller number of total steps. Interestingly, depending on the number of additional locations that sampling takes place in, the performance of both reinforcement learning agents varies drastically. Additionally, as we take more and more total walks, the gap between the DQN agent and the uniform sampling shrinks, likely due to the fact that both are fundamentally using the Monte Carlo WoS sampler with no additions.

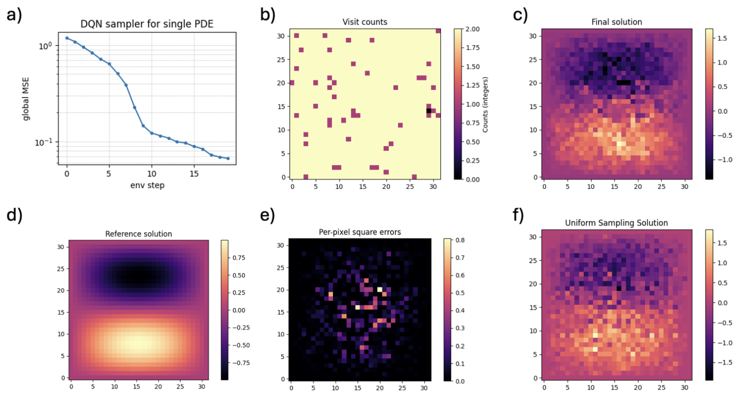


Figure 3: Example of DQN agent result. a) MSE for each iteration. b) Final visit counts (each visit = 10 walks) c) Final solution from the DQN agent d) Analytical reference solution e) Per-pixel error of final solution f) Uniform sampling solution (10 walks each)

6.2.2 Proximal Policy Optimization

The PPO-trained agent results are shown in the second line of Table 2. We notice that PPO constantly underperforms both the DQN agent as well as simple uniform sampling. A potential cause of this is due to PPO's clipping of the gradient function leading to less exploration. Due to the nature of the problem, it is likely that the optimal solution does involve some degree of uniform increase in visit

Method	MC Iterations	MSE	PSNR
WoS	100	0.0063	26.68
WoS	1000	0.00069	36.31
GAN Denoiser	10	0.0046	28.43
GAN Denoiser	100	0.00087	34.19
PPO + NN (v1)	153.89	0.00092	33.66
PPO + NN (v2)	90.63	**0.00045**	**37.51**
PPO + NN (v3)	10.53	0.0042	29.08

Table 3: Performance Metrics for RL-based sampling with hybrid Monte Carlo-Neural Network

counts with some additional samples for specific grid points here and there. This is shown in Figure 4 for two different PDEs, in which one plateaus at a certain MSE (due to constantly sampling the same points). However, the model ends up converging closer to the other two methods at higher iterations.

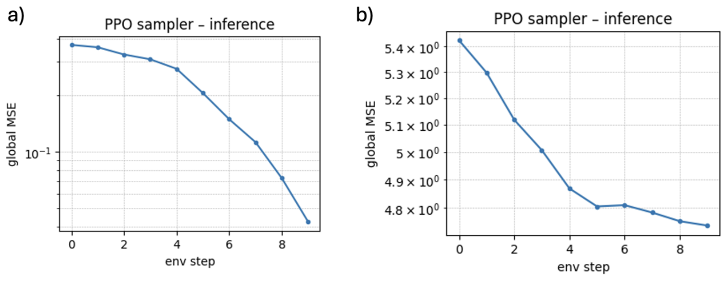


Figure 4: Example of PPO MSE curves for two different PDEs

6.3 Experiment 2

The experimental results for our second set of experiments are shown below.

6.3.1 v1

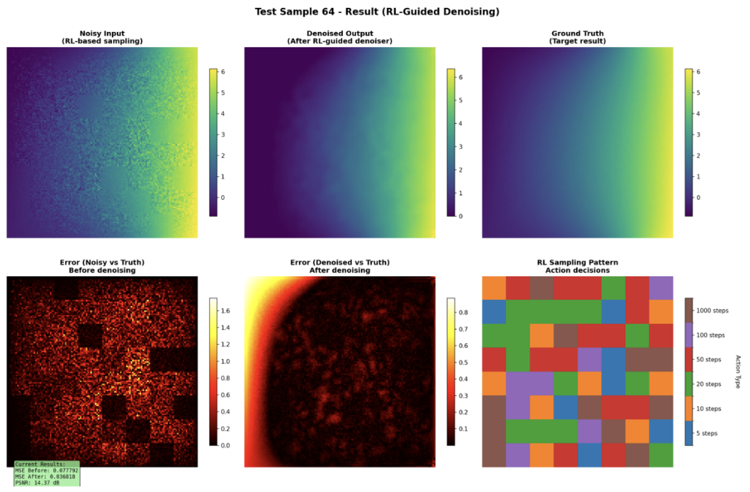
The results for v1 of our RL + MCNN setup are shown in Table 3 (top of the page). We see that training the RL agent and neural network denoiser in tandem with one gradient step each yields decent results as shown in the table. We show one sample test example in Figure 5. We see that the agent tends to choose a variety of different actions at each grid point, leading to pretty significant differences in variances of the input into the denoiser, which may impact its performance.

6.3.2 v2

The results are shown in the second-to-last line of Table 3. We see here that our agent tends to take mostly a small number of steps between 5-20 at most points, but samples a large amount from certain grid points. An example is shown in Fig. 6. The average number of steps taken for each PDE is still high, but lower than that of the v1 agent. Training the denoiser more at the end to "fine-tune" on the RL outputs led to a lower MSE and higher PSNR. This is likely because both the denoiser neural network weights were updated further based on the RL agent outputs, while also more training provides a more stable reward function for PPO.

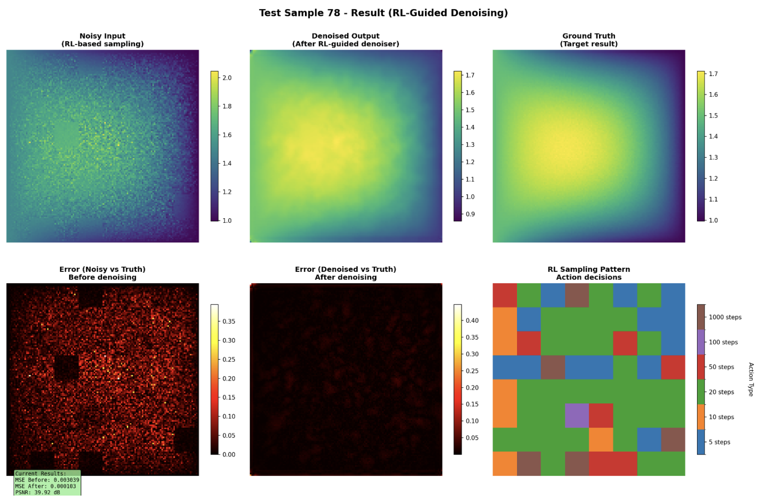
6.3.3 v3

The results are shown in the last line of Table 3. We see here that our agent tends to take a much smaller number of steps compared to the other two versions. This is likely because pre-training the neural network denoiser led to more stable reward updates at lower iterations, so the agent found a local optimum at around 10 iterations. It is also a lot more uniform. An example is shown in Fig. 7.



```
Current Results:
MSE Before: 0.077792
MSE After: 0.036818
PSNR: 14.37 dB
```

Figure 5: Example for PPO + NN (Denoiser) v1. Top right shows the noisy input as sampled based on the action map on the bottom right. Top middle shows the denoised output and top right shows the ground truth. Bottom left shows the errors before denoising, and bottom middle shows the errors after.



```
Current Results:
MSE Before: 0.003039
MSE After: 0.000103
PSNR: 39.92 dB
```

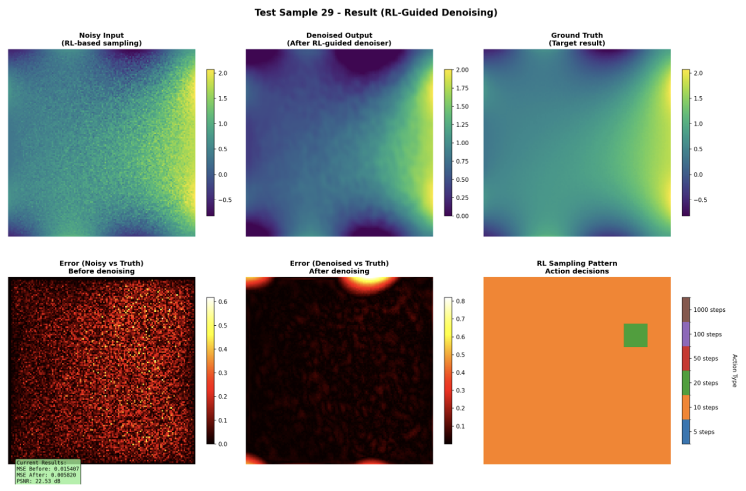
Figure 6: Example for PPO + NN (Denoiser) v2. Same layout at previous figure.

However, because we are taking much less steps, the denoiser has to "do more work," and we see that the final solution performance is worse than the previous two versions.

7 Discussion

In this study, we explored the integration of deep RL techniques into Monte Carlo-based PDE solvers with the aim of improving their efficiency and accuracy. We primarily focused on two areas: RL agents with adaptive sampling and hybrid RL agent-neural network denoiser approaches.

From our results, we see that RL agents can successfully adaptive sampling strategies that selectively increase sampling in areas of higher uncertainty/error, thereby reducing MSE quicker than uniform sampling. However, we note that this is more prominent under constrained sampling budgets, and these methods converge at larger sampling budgets. Interestingly, we saw that the implemented PPO algorithm consistently underperformed compared to both Deep Q-learning and uniform sampling. This underperformance can possibly be attributed to a few algorithmic mismatches with the overall



```
Current Results:
MSE Before: 0.015407
MSE After: 0.005820
PSNR: 22.53 dB
```

Figure 7: Example for PPO + NN (Denoiser) v3. Same layout at previous figure.

PDE solving environment. Our discrete action space and (to a high degree) deterministic optimal actions may lend itself to perform better with Deep Q-Learning, where the policy learns the Q-values for each action for a given state. The superior performance of uniform sampling over PPO further suggests that the policy gradient method struggled to learn effective exploration strategies. As seen in Fig. 3, in certain environments, the agent gets trapped in suboptimal stochastic policies that failed to identify high-uncertainty/error regions to sample from. Additionally, its possible that there are many hyperparameters that can be further tuned for PPO that may have drastic impacts on the resulting policy.

In our hybrid method, combining an RL-based sampling strategy with a neural network denoiser proved highly promising. Particularly in experiment v2—where the neural network was fine-tuned after RL training—the method achieved better solution accuracy with much fewer total Monte Carlo iterations than standard methods (90 vs 1000), such as uniform sampling or prior neural denoising techniques like GANs. This indicates that incorporating neural networks as a denoising step can significantly enhance the efficiency of adaptive RL-based PDE solvers. The success of this hybrid approach may come from both the RL agent's adaptive sampling as well as the denoiser itself. These two parts of the PDE solver are likely coupled together, with the denoiser learning how to effectively go from the output of the RL agent to the ground truth solution. This idea is also noted in [28]. By concentrating computational budget on regions that might have higher-uncertainty, the RL agent helps provide the denoiser a better signal-to-noise ratio overall for the entire grid, allowing it to be more effective. From a more theoretical perspective, these experiments showed that we are able to use learned priors (i.e. the RL agent) to increase efficiency in PDE solving and that there is potential in embedding reinforcement learning principles within more traditional methods.

Howevwer, we do note that there are a few limitations to this study. We constrained our action space to be 64 total actions for each PDE, with each action covering a 16x16 block. Additionally, we used simple 2D Poisson equations. This formulation may not generalize well to higher-dimensional PDEs or more complex geometries beyond squares. Additionally, we struggled a lot with tuning hyperparameters to improve the performance of PPO. In a sense it seem that much more of the performance gains are coming from the denoiser itself rather than the RL agent, although this would require more study. Finally, this overall end-to-end approach adds a lot more complexity to the problem of solving Monte Carlo PDEs. Each part of the system requires specific fine-tuning, and trying to combine the RL agent and a denoiser can lead to unstable results, on top of increased complexity compared to traditional methods.

8 Conclusion

Solving elliptic PDEs efficiently remains a challenge in computational graphics and simulations. In this work, we explore a new avenue of solving PDEs by using a combination of reinforcement learning techniques matched with neural networks. We show that RL has to potential to significantly improve sampling techniques and resulting solution quality, especially under limited computational resources. Our main contributions include introducing an adaptive sampling framework driven by RL techniques, as well as enhancing this approach by pairing it with neural-network based denoising techniques. We show that our RL + MCNN setup performs better than the 1000th iteration of regular Monte Carlo WoS solvers, while only costing around 90th iterations of Monte Carlo steps, which is a 10x decrease in computational budget.

With more time and computational resources, there are a lot more interesting avenues of research that can be explored on this topic. The first would be expanding the action space of our RL + Denoiser setup, particularly to the point where for each grid point, our RL agent can choose how many walks to take. The more fine-grained decisions may help reduce computational budget even further. Additionally, the inputs into the RL agent can be reworked to be similar to [28], where the action steps are much closer together. In our study, the action steps range from 5 to 1000, which may result in the agent choosing more steps than required. Lastly, it would be interesting to consider different reward structures other than just based off of MSE, as it may not be the goal of the RL agent to output a solution with the lowest MSE, but rather an output that works best with the denoiser.

Overall, these results demonstrate the immense potential of reinforcement learning as a powerful tool to augment the partial differential equation solving process. They are an effective way at training adaptive computational methods, which provides more flexible approaches to solving PDEs.

9 Contributions

Ivan worked alone on this project and carried out all experiments.

10 Acknowledgments

We received useful advice for structuring the project from Sri Jaladi, our CS224R mentor.

References

[1] Richard Bellman. 1957. *Dynamic programming*. Princeton University Press.

[2] Jens Berg and Kaj Nyström. 2018. A unified deep artificial neural network approach to partial differential equations in complex geometries. *Neurocomputing* 317 (2018), 28–41.

[3] Marcelo Bertalmio, Guillermo Sapiro, Vincent Caselles, and Coloma Ballester. 2000. Image inpainting. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), 417–424.

[4] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. 2010. *Polygon mesh processing*. CRC press.

[5] Susanne Brenner and Ridgway Scott. 2008. *The mathematical theory of finite element methods*. Vol. 15. Springer Science & Business Media.

[6] William L Briggs, Van Emden Henson, and Steve F McCormick. 2000. *A multigrid tutorial*. SIAM.

[7] Samuel N. Cohen, Deqing Jiang, and Justin Sirignano. 2023. Neural Q-learning for solving PDEs. arXiv:2203.17128 [math.NA] `https://arxiv.org/abs/2203.17128`

[8] Keenan Crane. 2013. *Digital geometry processing with discrete exterior calculus*. Carnegie Mellon University.

[9] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H Barr. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), 317–324.

[10] Howard Elman, David Silvester, and Andrew Wathen. 2014. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Oxford University Press.

[11] Víctor Elvira, Luca Martino, David Luengo, and Mónica F Bugallo. 2019. Advances in importance sampling. *Statist. Sci.* 34, 1 (2019), 1–23.

[12] Thomas JR Hughes. 2012. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation.

[13] Zichao Jiang, Junyang Jiang, Qinghe Yao, and Gengchao Yang. 2023. A neural network-based PDE solving algorithm with high precision. *Scientific Reports* 13, 1 (2023), 4479.

[14] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[15] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. 2020. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895* (2020).

[16] Zilu Li, Guandao Yang, Xi Deng, Christopher De Sa, Bharath Hariharan, and Steve Marschner. 2023. Neural Caches for Monte Carlo Partial Differential Equation Solvers. In *SIGGRAPH Asia 2023 Conference Papers* (, Sydney, NSW, Australia,) *(SA '23)*. Association for Computing Machinery, New York, NY, USA, Article 34, 10 pages. https://doi.org/10.1145/3610548.3618141

[17] Luca Martino, Víctor Elvira, David Luengo, and Jukka Corander. 2018. Adaptive importance sampling: The past, the present, and the future. *Statistical Methods & Applications* 27, 1 (2018), 1–29.

[18] Michael Mascagni and Ashok Srinivasan. 2004. *Monte Carlo methods for partial differential equations*. Springer.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[20] Mervin E. Muller. 1956. Some Continuous Monte Carlo Methods for the Dirichlet Problem. *The Annals of Mathematical Statistics* 27, 3 (September 1956), 569–589. http://links.jstor.org/sici?sici=0003-4851%28195609%2927%3A3%3C569%3ASCMCMF%3E2.0.CO%3B2-2

[21] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. 2019. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.* 378 (2019), 686–707.

[22] Pratik Rathore, Weimu Lei, Zachary Frangella, Lu Lu, and Madeleine Udell. 2024. Challenges in Training PINNs: A Loss Landscape Perspective. arXiv:2402.01868 [cs.LG] https://arxiv.org/abs/2402.01868

[23] Karl K Sabelfeld. 2013. *Monte Carlo methods in boundary value problems*. Springer Science & Business Media.

[24] Rohan Sawhney and Keenan Crane. 2020. Monte Carlo Geometry Processing: A Grid-Free Approach to PDE-Based Methods on Volumetric Domains. *ACM Trans. Graph.* 39, 4 (2020).

[25] Rohan Sawhney and Keenan Crane. 2020. Monte Carlo geometry processing: A grid-free approach to PDE-based methods on volumetric domains. *ACM Transactions on Graphics* 39, 4 (2020), 1–18.

[26] Rohan Sawhney, Dario Seyb, Wojciech Jarosz, and Keenan Crane. 2022. Grid-free Monte Carlo for PDEs with spatially varying coefficients. *ACM Trans. Graph.* 41, 4, Article 53 (jul 2022), 17 pages. https://doi.org/10.1145/3528223.3530134

[27] Rohan Sawhney, Dario Seyb, Wojciech Jarosz, and Keenan Crane. 2022. Walk on stars: A grid-free Monte Carlo method for PDEs with Neumann boundary conditions. *ACM Transactions on Graphics* 41, 4 (2022), 1–16.

[28] Antoine Scardigli, Lukas Cavigelli, and Lorenz K. Müller. 2023. RL-based Stateful Neural Adaptive Sampling and Denoising for Real-Time Path Tracing. arXiv:2310.03507 [cs.CV] `https://arxiv.org/abs/2310.03507`

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[30] Jonathan Richard Shewchuk. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational geometry* 22, 1-3 (2002), 21–74.

[31] Hang Si. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Software* 41, 2 (2015), 1–36.

[32] Jos Stam. 1999. Stable fluids. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), 121–128.

[33] John C Strikwerda. 2004. *Finite difference schemes and partial differential equations*. SIAM.

[34] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[35] Sifan Wang, Yujun Teng, and Paris Perdikaris. 2021. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing* 43, 5 (2021), A3055–A3081.

[36] Sifan Wang, Xinling Yu, and Paris Perdikaris. 2020. When and why PINNs fail to train: A neural tangent kernel perspective. *CoRR* abs/2007.14527 (2020). arXiv:2007.14527 `https://arxiv.org/abs/2007.14527`

[37] Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. 2022. Neural fields in visual computing and beyond. *Computer Graphics Forum* 41, 2 (2022), 641–676.

A Additional Experiments

A.1 Experiment 1

Here, we include another plot comparing all three strategies.

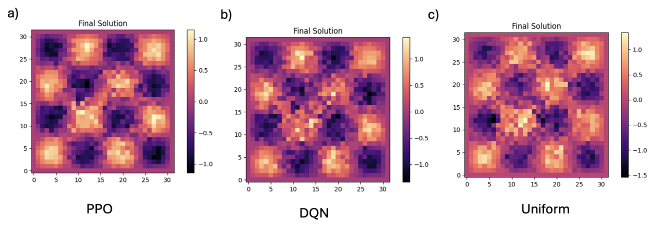


Figure 8: Additional result from adaptive sampling strategies showing that they converge

A.2 Experiment 2

Here, we include a few more plots showing the agent in different environments (each PDE essentially has its own environment). These are shown in Fig. 9 and 10, and 11.

It's also interesting to note that our entire setup struggles on certain PDEs, especially in our training dataset. One example is shown below in Fig. 12, where the denoiser (UNet) ends up worsening the output from the RL-driven Monte Carlo solver.

Furthermore, we also attach the rewards from training our agent models for v1 and v3 below. More gradient steps with the denoiser as well as pre-training lead to more stable reward updates. These are shown in Fig. 13 (v1) and Fig. 14 (v3).

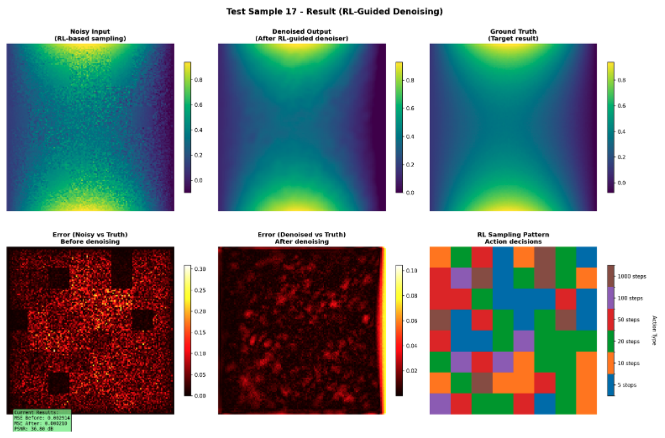


Figure 9: Additional result from v2

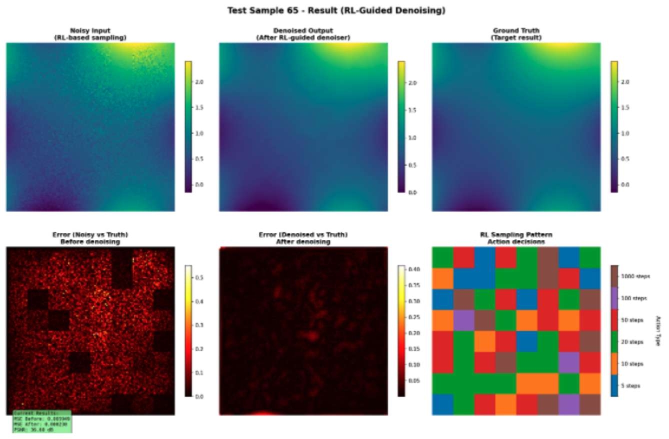


Figure 10: Another additional result from v2

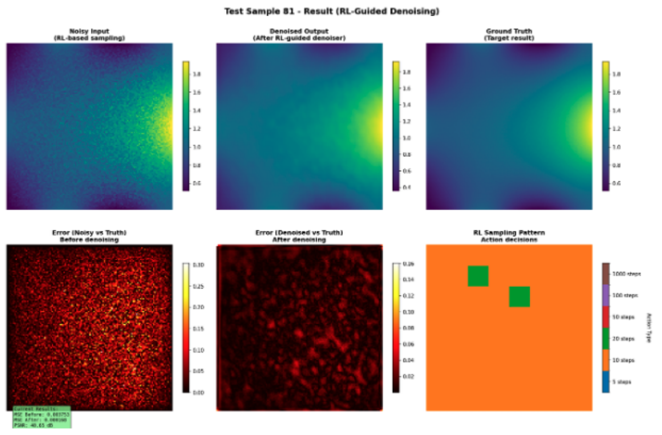


Figure 11: Additional result from v3

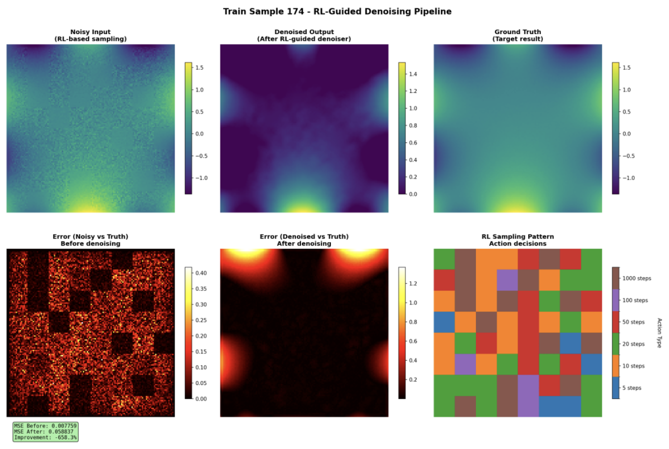


Figure 12: Bad result from train dataset at the end of training

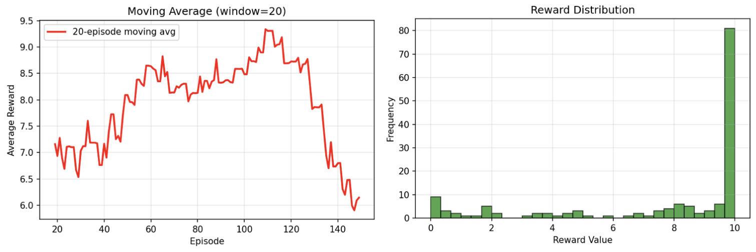


Figure 13: Rewards for v1

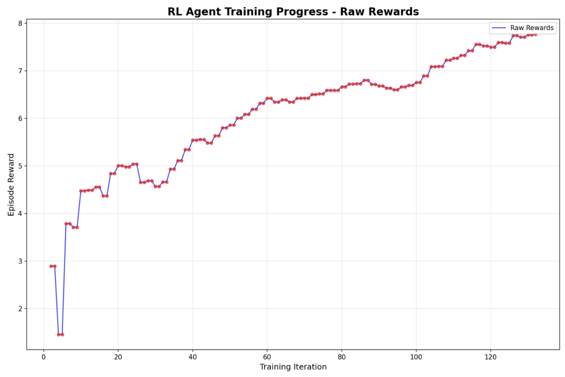


Figure 14: Raw reward graph for training v3

B Implementation Details

The architecture for the Deep Q-Learning agent consisted of a simple two-layer MLP where the hidden layer size is 128 with ReLU in between. For PPO, the agent consisted of two convolutional layers (32 and 64), following by a linear layer (256) and then two separate linear layers to split the output into policy and values. There are certainly improvements that can be made to this architecture, such as adding convolutional layers to the DQN. However, these were not explored for the scope of this project. The hyperparameters are further below in the paper. A full hyperparameter search was not conducted, so these values are likely not optimal for maximal performance.

The architecture for the RL-based sampling with hybrid Monte Carlo-Neural Network agent consisted of a convolutional architecture for the RL-agent to output a grid of sampling locations. The specific architecture is shown in Fig. 15. For the denoiser, the architecture is directly adopted from [28], which is based on the OIDN denoiser by Intel. Since then, more powerful denoisers heave likely more been developed and are more performant.

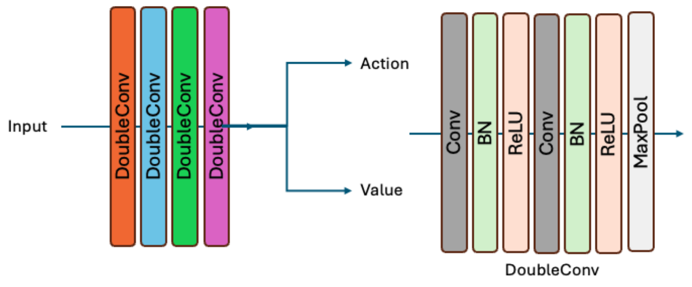


Figure 15: Agent architecture for Experiment 2

RL Environment for Experiment 1

State Space

- Shape: $(H, W, 3)$ Three-channel observations:
  1) **Channel 1**: Current Monte Carlo estimate $\hat{u}$
  2) **Channel 2**: Residual map $|\Delta\hat{u} - f|$
  3) **Channel 3**: Normalized visit count (sampling density)

Action Space

- $\text{Discrete}(N)$, where $N = H \times W$ (number of grid points)
- Each action selects K grid cells to refine with additional Monte Carlo samples

Hyperparameter	**PPO**	**DQN**
Learning rate (lr)	3×10^{-4}	3×10^{-4}
Discount factor (γ)	0.99	0.99
GAE lambda (λ)	0.95	—
Clip epsilon (clip_eps)	0.2	—
Entropy coefficient (ent_coef)	0.1	—
Value-function coefficient (vf_coef)	0.5	—
Epsilon (ε-greedy)	—	0.1

For the DQN, to encourage more exploration and avoid the policy repeatedly sampling from the same points at times, we also experimented with

$$Q_{adjusted}(s, a) = \frac{Q(s, a; \theta)}{0.1 + \sqrt{count(a)}}$$

to force more exploration. However, experiments ended up yielding similar results. The loss is given by

$$\mathcal{L}_{DQN}(\theta) = \mathbb{E}_{(s,\mathbf{a},r,s')\sim\mathcal{D}}\left[\frac{1}{K}\sum_{k=1}^{K}\Big(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a_k; \theta)\Big)^2\right]$$

where a_k is the entire action vector with K different locations. Each batch of actions is given the same reward signal for faster computation, but more optimal methods would have per-pixel rewards.

For the PPO, the relevant equations follow [29], except the actions were samples as follows

$$\mathbf{a}_t = \text{TopK}\left(\tilde{\pi}(\cdot|s_t), K\right)$$

However, as opposed to the DQN method, each action is treated separately with its own per-pixel MSE reward for better signal.

RL Environment for Experiment 2

State Space

- Two-channel input of shape $(2, 128, 128)$:
  1) **Channel 1**: First iteration result (initial noisy solution)
  2) **Channel 2**: Variance map (difference between 1st and 5th iterations)

Action Space

- Discrete (6): sampling decisions per 16×16 block
- A 128×128 image divided into $8 \times 8 = 64$ blocks of size 16×16 pixels each
- Actions: 5 steps (very low sampling); 10 steps (low sampling); 20 steps (medium-low sampling); 50 steps (medium sampling); 100 steps (medium-high sampling); 1000 steps (high sampling)
- The agent makes 64 sequential decisions (one decision per block)

PPO hyperparameters:

Hyperparameter	**Value**
`train_batch_size`	64
`sgd_minibatch_size`	16
`num_sgd_iter`	5
`learning_rate`	2×10^{-4}
γ	0.99
λ	0.95
`clip_param`	0.2
`entropy_coeff`	0.02
`vf_loss_coeff`	0.5
`denoiser_lr`	1×10^{-4}

Denoiser training hyperparameters:

Hyperparameter	**Value**
`denoiser_epochs`	200
`denoiser_lr`	1×10^{-4}
`batch_size`	8
`weight_decay`	1×10^{-4}
Learning rate scheduler	`CosineAnnealingWarmRestarts`