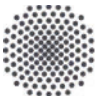


University of Stuttgart

Institute for Artificial Intelligence
Machine Learning for Simulation Science

Universitätsstraße 32
70569 Stuttgart

Bachelor Thesis

Stochastic Query Synthesis for Neural PDE Solvers

Finn Ullrich

Study program:	Informatik, B. Sc.
1. Examiner:	Prof. Dr. Mathias Niepert
2. Examiner:	
Advisors:	Daniel Musekamp, M.Sc.
start date:	28.10.2024
end date:	28.04.2025

Abstract

PDEs are highly influential in physics and are describing various phenomena in the world, from wave movement to electro-magnetics. The problem arises when one tries to solve them, which requires enormous computing power for a numerical solution. To overcome the limitiations, neural PDE solvers have been proposed, using neural networks to approximate the solution trajectories. However, neural PDE solvers require training data from an computationally expensive numerical solver. Therefore, Musekamp et al. created a benchmark, which investigates active learning for neural PDE solvers. Active learning can reduce the amount of data required, while keeping the same performance. In this work, we will demonstrate a new strategy of selecting samples called stochastic query synthesis. Following this, we will remove the pool currently used and rather create a Markov chain directly from the input space containing unlabeled instances. The transition probability is based on the unadjusted Langevin algorithm, allowing us to sample by exploiting gradient information. To retrieve a better result, instead of just one chain, we will create multiple parallel chains, and only take the last state as input. We will show that this approach is equally effective as the currently implemented pool-based implementation. However, there are still performance problems that need to be solved in the future, to make it viable in practice.

Kurzfassung

PDGs sind einflussreich in der Physik und beschreiben verschiedene Phänomene in der Welt, von der Wellenbewegung bis zur Elektromagnetik. Problematisch wird es, wenn man versucht, sie zu lösen. Um eine numerische Lösung zu erhalten, ist eine enorme Rechenleistung erforderlich. Um diese Einschränkungen zu überwinden, wurden neuronale PDG-Löser vorgeschlagen, die neuronale Netze verwenden, um die Lösungstrajektorien zu approximieren. Neuronale PDG-Solver benötigen jedoch Trainingsdaten von einem rechenintensiven numerischen Solver. Daher haben Musekamp et al. einen Benchmark entwickelt, der aktives Lernen für neuronale PDG-Löser untersucht. Aktives Lernen kann die Menge der benötigten Trainingsdaten bei gleicher Leistung reduzieren. In dieser Arbeit demonstrieren wir *stochastic query synthesis* als eine neue Strategie zur Auswahl von Stichproben. Anstelle des aktuell verwendeten Pools wird eine Markow-Kette aus dem Eingabebereich mit unbeschrifteten Instanzen erzeugt. Die Übergangswahrscheinlichkeiten werden durch den *unadjusted Langevin algorithm* definiert, was uns erlaubt, Stichproben unter Ausnutzung der Gradienteninformation zu ziehen. Um dieses Ergebnis zu verbessern, erstellen wir mehrere parallele Markov-Ketten, indem wir nur den letzten Zustand als Eingabe verwenden. Wir werden zeigen, dass diese Art der Implementierung genauso effektiv ist wie die derzeit verwendete Pool-basierte Implementierung. Es gibt jedoch noch Performance-Probleme, die in Zukunft gelöst werden müssen, um den Algorithmus effizient einsetzen zu können.

Contents

List of Figures

List of Algorithms

1 Introduction

In deep learning, a key component is the training set from which our model can learn [LBH15]. It consists of the input data and the associated label. For example, if we want our model to recognise pictures of dogs and cats, the input data would be the pictures, and the associated labels would be the classification as dog or cat. Usually, the amount of training data is required to be quite large to keep the deviation between the expected output of our model and the actual output small. However, the process of labeling data is not free. Often the computational cost is many times greater than the cost of generating the input itself [Set09]. A common technique to reduce the number of training samples required while still achieving high accuracy is *active learning*. Rather than just randomly labeling each piece of input data and using it as training data, we instead apply an algorithm that selects the samples that give us a high amount of information. This has been applied to various fields, e. g. drug discovery [Rek19] or text categorisation [Lew95]. The field we are intrested in for this work is active learning for neural PDE solvers.

The solution to a PDE is typically approximated using a numerical solver, which is both resource-intensive and time-consuming. To reduce the effort, neural PDE solvers have been introduced [LVP+23], which require a numerical solver for their training data and pass these results on to the neural network. Here we still have the problem of the high cost of labeling by numerical solvers. Therefore, pool-based active learning for neural PDE solvers was introduced by Musekamp et al. [MKH+24] and applied to different neural surrogate models.

Pool-based methods are one of the most common strategies for active learning because they give good results and are easy to implement. However, the creation of a pool from the input space has its drawbacks. In particular, since the pool size is determined by a hyperparameter, the challenge is to find the correct pool size. If the pool is too small, the creation of the pool is fast, but important regions with high sample information from the input space will be omitted. If the pool is too large, it is too expensive to build.

Instead, we can sample directly from the input space, known as query synthesis [Ang88]. The main advantage is that we can exploit the gradient of uncertainty by sampling from the largest increase. Unlike pool-based methods, which can become trapped in local optima, query synthesis allows a more global search for informative samples. This is also the main goal of this thesis: to replace the old pool-based method with a query synthesis strategy using the uncertainty of the model as measurement for sample information. More specifically, we want to utilize gradient information of the uncertainty.

The gradient information allows us to efficiently sample from the complete underlying distribution, or push the sampling towards higher uncertainties. This should be reducing the deviation between actual and expected results compared to the pool-based method, while improving the computational speed.

We begin with a brief introduction to the various fundamentals necessary to understand the current implementation. Next, we describe the methodology, followed by an explanation of the experiments and their implementation. Also we will have a comparision to similar already existing work. We then discuss the results. Finally, we conclude with a summary of our findings and discuss possible directions for future research.

2 Foundations

We begin by outlining the basic requirements for understanding this work. This includes all essential concepts related to active learning for neural PDE solvers. First, we will explain what a partial differential equation (PDE) is and why solving PDEs is important. Next, we will introduce autoregressive neural PDE solvers and discuss how they work. Finally, we will explain the basic principles of active learning and go through more detailed variations used in this work.

2.1 Partial Differential Equations

Since this paper is about solving partial differential equations (PDEs) with neural networks, it is important to understand what exactly a PDE is. We have a D-dimensional spatial domain $\mathcal{X}$, $x = [x_1, x_2, \dots, x_D]^\top \in \mathcal{X}$, and a temporal domain $t \in [0, T]$ (Brandstetter et al., 2022):

$$\partial_t u = F(\lambda, t, x, u, \partial_x u, \partial_{xx} u, \dots), \qquad (t, x) \in [0, T] \times \mathbb{X} \tag{2.1}$$

$$u(0, x) = u^0(x), \qquad \mathcal{B}[u](t, x) = 0 \qquad x \in \mathbb{X}, (t, x) \in [0, T] \times \partial\mathbb{X} \tag{2.2}$$

We want to know the unique solution $u : [0, T] \times \mathbb{X} \to \mathbb{R}$ with the initial conditions (IC) $u(0, x) = u^0(x)$ and its boundary conditions $\mathcal{B}[u](t, x) = 0$. The PDE parameters are represented by the vector $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_l) \in \mathbb{R}^l$ with $\lambda_i \in [a_i, b_i]$ and influence the underlying system. The initial conditions describe the initial state of the system, while the boundary conditions define the PDE at the definition boundaries. Both conditions are necessary to obtain a unique solution. PDEs play an important role in the modern world of physics, for example in the heat equation, which describes how heat is distributed in an area over time:

$$\lambda \partial_t u = \Delta u \tag{2.3}$$

with $\Delta u = \sum_{i=1}^{d} \partial_{x^i x^i} u$, in d-dimensions.

2.2 Autoregressive Neural PDE solvers

Neural PDE solvers can be broadly divided into two categories: Neural operator methods and autoregressive methods. Neural operator methods aim to learn a global mapping from input conditions (e.g. initial conditions and PDE parameters) to the entire solution trajectory. These models are typically designed to generalise across different resolutions, geometries or boundary conditions. Autoregressive methods, on the other hand, follow a time marching approach, solving PDEs iteratively by advancing the solution one time step at a time. The process of repeatedly

feeding the output state as input for the next step is called rollout. In this setting, the model learns a temporal forward operator $\mathcal{A}$ so that the system evolves according to

$$u(t + \Delta t) = \mathcal{A}(\Delta t, u(t)) \tag{2.4}$$

where $\mathcal{A} : \mathbb{R}_{>0} \times \mathbb{R}^N \to \mathbb{R}^N$ acts as a learned surrogate for the underlying dynamics [BWW22]. This autoregressive formulation is particularly suitable for time-dependent PDEs.

2.3 Active learning

Before we explain active learning - a subfield of machine learning - we have to explain how machine learning itself works. In the words of Ayodele (2010): "Machine Learning can still be defined as learning the theory automatically from the data, through a process of inference, model fitting, or learning from examples".

But what exactly is "the data"? There are different strategies how to learn which also define how our machine model learns with data, but we will keep our focus on supervised learning.

In supervised learning, a model is trained on a training data set consisting of input-output pairs. The input data represents the information we want the model to learn from or make predictions about, while the output data - known as the label - is the expected result for each input. Although we often use the term "classification", labels are not limited to discrete categories. If the output is a continuous value, the model is performing regression instead of classification. From the whole input space, the samples that get added to the training set are chosen at random [Ayo10].

In active learning, we try to minimize the amount of training samples we use as input, while keeping the same/better learning curve. This is done by activley choosing the samples for our training set [RSKB17]. This means we activley query our input data to get it labeled, therefore getting an output value, by a so called oracle. An oracle is the general term for something that can always label correctly our sample we queried [Set09]. If we train our ensemble, which has multiple neural networks, the approximation for the solution of a sample can differ between the models. There are mainly two factors responsible for this behaviour. Firstly, the models are using stochastic methods, specifically gradient descent, to approximate solution trajectories, which can lead to different results with the same input. Secondly, the initial weights inside the neural network are stochastically created and are therefore different between models. The variance between them can give us an indication about how informative the sample will be. The informativeness therefore called "uncertainty" and is modeled via a probabalistic function, with a Query-by-Committee algorithm (Seung et al. 1992):

$$a_{QbC}(\psi_i) := \frac{1}{N_t N_x N_c} \sum_{j=1}^{N_t} \sum_{k=1}^{N_x} \frac{1}{N_m} \sum_{m=1}^{N_m} ||\hat{u}_{i,m}(t_j, x_k) - \overline{\hat{u}_i}(t_j, x_k)||_2^2 \tag{2.5}$$

We have now established a way to define the information content of a sample, but we have yet to explore how samples are actually queried in active learning. Broadly speaking, active learning employs three main strategies, which are illustrated in a simplified manner in Fig. 2.1.

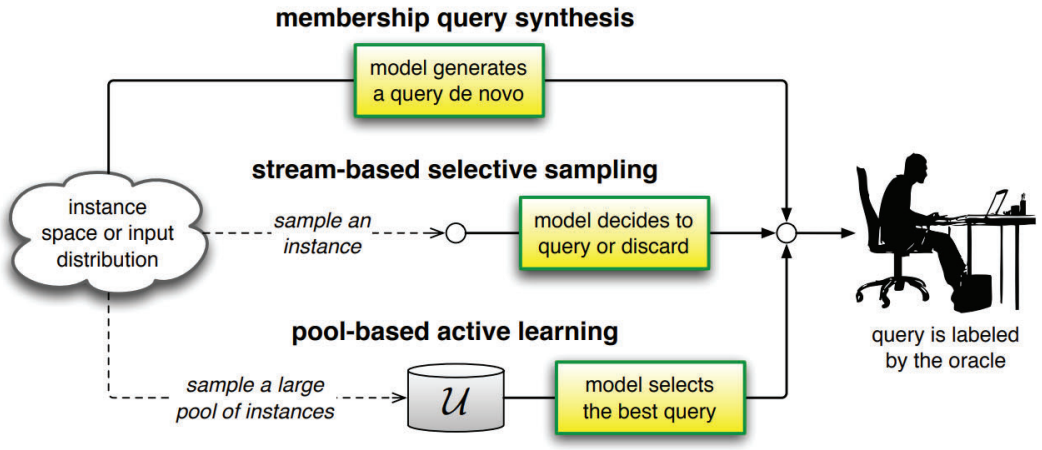


Figure 2.1: The three main methods of active learning by Settels [Set09]

.

In pool-based active learning, a collection of samples is first drawn from the input space. The learner then selects specific samples from this pool to query the oracle. In contrast, membership query synthesis (in this work referred to as stochastic query synthesis) bypasses the predefined pool altogether. Instead, the learner queries directly from the input space - or creates completly new unlabeled instances to query.

Finally, in stream-based selective sampling, samples arrive sequentially from the input space. The learner must decide in real-time whether to query the oracle for each incoming sample or discard it. Since the data stream may come from a non-stationary distribution, this approach can sometimes resemble the behavior of query synthesis.

2.3.1 Pool-based active learning

The pool-based approach is about sampling a large pool of unlabeled instances from the original instance space/input distribution. Samples from this pool then get queried to an oracle which will generate a label, visualized in Fig. 2.2.

In mathematical terms, we will reuse the definition from Ren et al. [RXC+21]. Let $U^n = \{\mathcal{X}, \mathcal{Y}\}$ be the unlabeled dataset with n samples, with $\mathcal{X}$ being the sample size and $\mathcal{Y}$ being the label size. P(x,y) is a potential distribution, with $x \in \mathcal{X}$, $y \in \mathcal{Y}$. It represents the joint probability distribution of the sample x and the label y. $L^m = \{X, Y\}$ is the training set with m labeled instances. In our case, the input space includes both ICs and PDE parameters for a given PDE, each taking continuous values. As a result, the input space is effectively infinite and our pool is created by sampling a finite amount of samples.

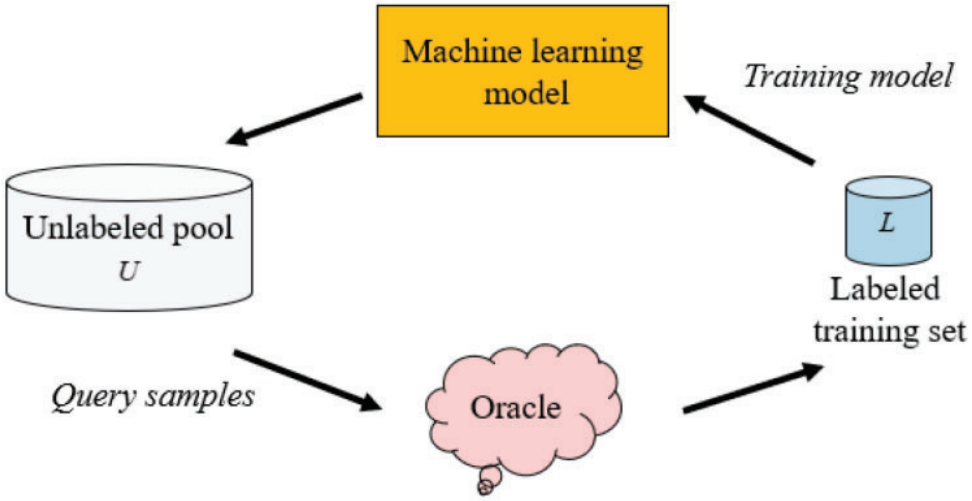


Figure 2.2: The pool-based cycle done by Ren et al. [RXC+21]

Now the goal is to create a query strategy Q that queries samples from U^n to be labeled by the oracle: $U^n \xrightarrow{Q} L^m$. We therefore have to choose which samples from we want to pass to the numerical solver and add to the training set. Here we can use the uncertainty as an indicator of the informativeness of the samples to select the samples. This approach is easy to implement and often improves on random sampling. It is the most common approach [ZLLC21]. In the context of deep active learning, the combination of deep learning and active learning, single-sample-based methods reach their limits as single points do not have a statistically significant impact. Furthermore, training on individual samples requires a full iteration of the deep learning network for each one, which is computationally expensive. To mitigate this, rather than querying single entities, it is more efficient to process an entire batch at once [SS17].

2.3.2 Batch active learning

When training a deep-learning network with a pool-based approach, it is inefficient to query single points. In batch active learning, instead of a single point, a whole batch of size K with $K \in \mathbb{N}$ is selected from the given pool [RXC+21]. The batch of unlabeled data $\mathcal{B} = \{x_1, x_2, ..., x_b\} \subseteq U$ is selected based on an acquisition function a. One common method how to generate a batch is to query the K samples with the highest uncertainty, also known as Top-K. The uncertainty is thereby evaluated at each sample inside the pool.

$$a_s^{\text{batch}}(\mathcal{I}^{\text{train}}; K) := \arg\max_{I \subseteq \mathcal{I}^{\text{pool}}, |I|=K} \sum_{i \in I} s(i; \mathcal{I}^{\text{train}}) \tag{2.6}$$

with a being the acquisition function, $\mathcal{I}^{\text{train}}$ being the indices of the samples in the training data set and $\mathcal{I}^{\text{pool}}$ being the indices of the samples in the pool data set [KFA+21].

2.3.3 Stochastic batch active learning

Top-K selection based on uncertainty alone can result in very similar samples, thereby reducing diversity. This, in turn, may reduce the information contribution of each selected sample, potentially hindering the learning efficiency of the model. Therefore Kirsch et al. [KFA+21] introduced stochastic batch active learning (SBAL). SBAL samples without replacement the input ψ from the pool U^n with the probability distribution

$$p_{\text{power}(\psi)} \propto \alpha(\psi)^m \tag{2.7}$$

with m being a hyperparameter, that is controlling the sharpness of the distribution.

2.3.4 Query synthesis active learning

Creating and evaluating the pool can be quite costly, especially if the pool size is set too large. To avoid this problem, another active learning method "Query synthesis" was introduced [Ang88]. Query synthesis samples to the oracle directly from the input space or can even create new samples to query [WHYL15].

For us this means to query samples directly to the numerical solver. This can be a big advantage for us, because we can potentially sample from regions with higher uncertainties than would be possible with pool-based methods. This is due to the fact that we are not constrained to a certain fixed number of samples, which could leave out potential sample areas with a higher uncertainty. An additional benefit for us is the ability to calculate the gradient of uncertainty, allowing us to sample from regions with the highest uncertainty.

The challenge here is to choose a fitting algorithm, defining the query strategy. We will choose an implementation of Markov chain Monte Carlo.

3 Method

In this chapter we want to discuss the method we want to implement and explain what exactly we are sampling from. First, we will give an overview of the current framework, which is important to understand what inputs we receive and where we send the output. Then we dive deep into the basic class of Markov chain Monte Carlo. The explanation of the basic class is followed by the specific implementation that we will be doing.

3.1 Active learning for PDEs

Before we go into more detail about what we are trying to change and implement, we first define the problem scope we are currently in. Our work builds upon and extends the AL4PDE benchmark framework introduced by Musekamp et al. (2025). We first examine the existing AL cycle, which is visualized in Fig.3.1.

The active learning process begins with the specification of a target partial differential equation (PDE) from which data is to be sampled. Once the specific PDE has been selected, the corresponding input space is constructed. This input space consists of samples from the domain of PDE parameters and initial conditions (ICs).

To begin the active learning cycle, a small set of random samples is randomly selected from the input space passed as input for the numerical solver, here called "PDE Solver". The PDE Solver computes the solution trajectories, which are used to initialize the neural surrogate model. The starting point is important to estimating the initial uncertainty assosiated with the model's predictions.

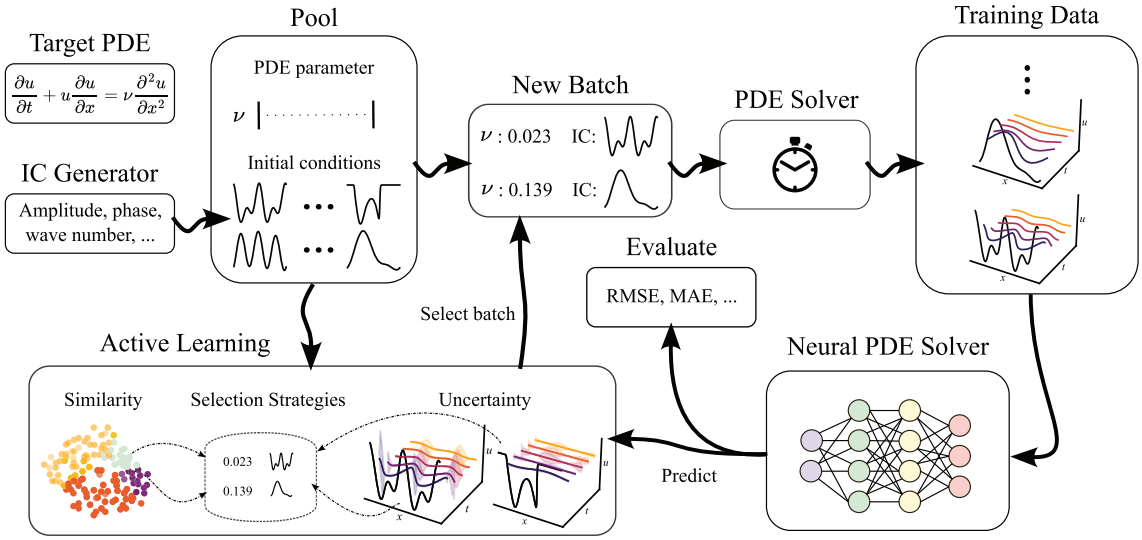
The uncertainty for each sample in the pool is quantified using the acquisition function a_{QbC}, based on the variance across different predictions across different models.

$$a_{QbC}(\psi_i) := \frac{1}{N_t N_x N_c} \sum_{j=1}^{N_t} \sum_{k=1}^{N_x} \frac{1}{N_m} \sum_{m=1}^{N_m} ||\hat{u}_{i,m}(t_j, x_k) - \overline{\hat{u}_i}(t_j, x_k)||_2^2 \tag{3.1}$$

where $\hat{u}_{i,m}$ denotes the prediction of the m-th model in the ensemble, and $\overline{\hat{u}_i}$ is the ensemble mean for input sample ψ_i evaluated over all time t_j and space x_k. The underlying distribution of the uncertainty is not known completly, it rather can be evaluated pointwise. Following this, a pool is constructed, based on random combinations of ICs and PDE parameters. Each sample inside the pool is not labeled and the solution is not known, but the uncertainty can be evaluated using the acquisition function from above.

In the next step, a batch of samples is selected from the pool using the SBAL strategy.

Figure 3.1: Active learning cycle from Musekamp et al. [MKH+24]



Once the batch is selected, it is removed from the pool and passed to the numerical PDE solver, to retrieve the label. The labeled samples are added to the training set, which is used as input for the neural PDE solver.

The updated model is again evaluated over the remaining pool samples, and the uncertianty is recalculated. The updated uncertainty is used as the new base for the next batch to select.

The query synthesis method we are implementing therefore is located between the input space and the neural PDE solver. Since we do not need a pool in query synthesis methods, it is removed. However we will still use the uncertainty as measure for the sample informativeness.

3.2 Target sampling distribution

We now take a closer look at the target distributions we aim to sample from. In the case of pool-based active learning with uncertainty as the target, we sample from an underlying distribution based on the evaluated uncertainty of all points in the pool. If we query a batch using SBAL, for example:

$$p_{\text{power}(\psi)} \propto \alpha(\psi)^m \tag{3.2}$$

ψ are the samples inside the pool. If the pool would not be uniformly created, a factor of $p_{pool}(\psi)$ would be added to SBAL.

In contrast, stochastic query synthesis removes the pool entirely, which also changes the nature of the target distribution. Instead of relying on a finite pool, we consider the uncertainty across the entire input space and generate queries accordingly.

3.3 Markov chain Monte Carlo

If we want to sample directly from the input space, we must somehow determine which samples to sample. In the best case we can query samples from higher uncertainties more often than those with low uncertainty. Since the uncertainty is not known continuously, we can not sample from it directly. Here we can exploit the class of algorithms called Markov chain Monte Carlo (MCMC). MCMC creates a chain of states, where each state is dependent on the previous state, and the whole chain is converging to a target distribution π. Effectivly meanining the difference between the samples created through the chain and samples created by directly sampling from π converges to zero.

MCMC starts with an initial sample X_1. The marginal distribution is called the "initial distribution" [Gey11]. Each subsequent sample is drawn based on the previous one using a transition probability distribution. Since the amount of samples is deterministic, this creates a discrete time Markov chain. Crucially, in a Markov chain, the next state X_{n+1} depends only on the current state X_n, rather then on the full history of the chain.

If we want the chain to converge to π, we have to understand what it means for a distribution to remain unchanged as the chain evolves, which leads us to the concept of stationary. Stationary means, that the conditional distribution for a subset of the chain does not depend on where the subset is taken from. We can be sure that a chain is stationary by checking if it is reversible. For a reversible chain, the distribution for $X_{n+1}, X_{x+2}, \ldots, X_{x+k}$ and $X_{n+k}, \ldots, X_{n+2}, X_{n+1}$ are the same.

Often a burn-in is used, i.e. the first n values of the chain are discarded with the aim of throwing away these samples before the chain converges. According to Geyer [Gey11], burn-in is unnecessary; starting from a reasonable point - "any point you don't mind having in a sample is a good starting point" - is just as effective. He goes on to explain that a common best practice is to use the end point from a previous run as the starting point for the next run. However, in our case, because the underlying distribution changes between iterations, we chose to randomly select the starting point. As long as the random sample is not an outlier - and we intend to train our model on this sample - the original guideline still applies. Therefore, it is omitted from all examples in this paper.

We still have the problem open on how we want to select the transition probability.

3.4 Undajusted Langevin algorithm

As a natural candiate for the transition probability is Langevin diffusion. The Langevin diffusion is defined as:

$$dL_t = dW_t + \frac{1}{2}\nabla \log \pi(L_t)dt \tag{3.3}$$

where W_t is the standard Brownian motion and π is the target density. Note that π must be continuously differentiable and non-zero, to ensure a well-defined gradient of the log-density $\nabla \log \pi$.

Intuitively, this seems like a good proposal choice, since L_t has π as stationary measure under suitable conditions.

If we want to use the diffusion as transition probability, we have to apply the Euler-Maruyama discretisation:

$$X_{n+1} = N(X_n + \frac{1}{2}h\nabla \log \pi(X_n), hI_k). \tag{3.4}$$

where h is the step size. The usage of Langevin diffusions as transisiton probability was first introdused by Parisi (1981), and further studied by Grenander and Miller (1994) as well as Roberts and Tweedie (1996). It is commenly referred to as the "unadjusted Langevin algorithm".

However, despite the continuous-time Langevin diffusion having π as its invariant distribution, the discretized version does not necessarily preserve this property. As a result, the Markov chain generated by ULA may either converge to an incorrect target distribution or exhibit slow convergence.

In our case, this could be an advantage. ULA may not converge to π, but due to the construction of the probability distribution, it will sample in the direction of the maximum point of π. Therefore, we will implement a version of MCMC with ULA as transition probability and call this method single-chain-ULA (SC-ULA). We will also implement a modification of SC-ULA. Since the states sampled from higher uncertainties are more valuable for us, we instead only take the last state as result. To match the amount of samples from SC-ULA, multiple chains are started in parallel. We will call this changed version parallel-chains-ULA (PC-ULA).

Both methods focus on sampling in the direction of maximum uncertainty, which can lead to a lack of diversity in the samples. To overcome this, we can modify the approach so that instead of always sampling towards the point of maximum uncertainty, it samples from the entire uncertainty. This can be achieved by combining the ULA with the Metropolis-Hastings algorithm.

3.5 Metropolis-Hastings algorithm

The Metropolis-Hastings algorithm is one possible implementation of constructing the Markov chain in MCMC and is developed by Hastings [Has70]. The algorithm is a further developlment of the Metropolis algorithm, which was introduced by Metropolis et al. [MRR+53]. Let $\pi(x)$ be the density we want to take samples from, called the target density. Then, the algorithm requires a distribution proportional to the target denisity, $\pi(x) \propto \tilde{\pi}(x)$ and a proposal kernel $q(X_{t+1}|X_t)$. The proposal kernel is a probabalistic function, and is proposing a new state X_{t+1} based on the old state X_t.

The transition from the Markov state X_t to X_{t+1} is then defined as follows: Sample a new State X_{t+1} from $q(X_{t+1}|X_t)$. The acceptance probability is then defined as:

$$\alpha(X_t, X_{t+1}) = \begin{cases} \min\left\{\frac{\tilde{\pi}(X_{t+1})q(X_t|X_{t+1})}{\tilde{\pi}(X_t)q(X_{t+1}|X_t)}, 1\right\} & \text{if } \tilde{\pi}(X_t)q(X_{t+1}|X_t) > 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.5}$$

The proposal is therefore accepted with the probabilty $\alpha(X_t, X_{t+1})$. This is known as Metropolis-Hastings step and if accepted, then set $X_{t+1} = X_{t+1}$, otherwise $X_{t+1} = X_t$ and restart the process. Importantly this also means if the sample is rejected, the sample before is twice in the chain.

For our implementation, we can draw an initial sample at random, to start the algorithm. For an implementation missing is still the proposal kernel. If our propsal kernel is equal to our target density, the acceptance rate would be one and we would converge in just one step. In the reality this is not possible, hence we have to approach this in a different way.

The easiest way for a creation of the kernel is to use Random Walk Metropolis (RMW). The proposals are drawn with i.i.d and a fixed symmetric denisity Z_i, e.g. $Z_i \sim N(0, \sigma^2 I_d)$. Each new proposal therefore has the form $X_{i+1} = X_i + Z_{i+1}$ [Ros+11].

This approach does work, however with a slow convergence rate [Nea+11], but more importantly, it does not utilize information we have. We have access to the gradient of the uncertainty, and for a faster convergence rate, we want to take advantage of it. Therefore we are searching for a proposal kernel, which also includes gradient information.

3.5.1 Metropolis adjusted Langevin algorithm

Since ULA alone is not necessarily converging to the correct distribution, it can rather be used as proposal kernel for Metropolis-Hastings. It takes advantage of the Metropolis-Hastings step, while keeping the discretised Langevin diffusion as a proposal. The next suggestion is therefore fetched with:

$$q(X_{n+1}|X_n) = N(X_n + \frac{1}{2}h\nabla \log \pi(X_n), hI_k) \tag{3.6}$$

then compute the acceptance probability:

$$\alpha(X_t, X_{t+1}) = \begin{cases} \min\left\{\frac{\tilde{\pi}(X_{t+1})q(X_t|X_{t+1})}{\tilde{\pi}(X_t)q(X_{t+1}|X_t)}, 1\right\} & \text{if } \tilde{\pi}(X_t)q(X_{t+1}|X_t) > 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.7}$$

To make it more understandable how we are going to implement it, we will first transform the formulas. We can rewrite q as following [RR98]:

$$q(X_{n+1}|X_n) = \frac{1}{2\pi\epsilon} \cdot \exp\left[\frac{-1}{2\epsilon}||X_{n+1} - X_n - \frac{\epsilon}{2}\nabla \log \pi(X_t)||_2^2\right] \tag{3.8}$$

Instead of calculating q like this, we will firstly compute the natural logarithm, and since we are using a ratio, we can leave out the constants. Doing all of this will change it to:

$$q(X_{n+1}|X_n) - \frac{1}{\epsilon} \cdot \sum (X_{n+1} - X_n - \frac{\epsilon}{2}\nabla \log \pi(X_n))^2 \tag{3.9}$$

If we still want to calculate α correctly, we will look at the logarithm of α:

(3.10)
$$\log \alpha = \log\left(\frac{\tilde{\pi}(X_{t+1})q(X_t|X_{t+1})}{\tilde{\pi}(X_t)q(X_{t+1}|X_t)}\right)$$

(3.11)
$$\log \alpha = (\log \pi(X_{t+1}) + \log q(X_t|X_{t+1})) - (\log \pi(X_t) + \log q(X_{t+1}|X_t))$$

4 Implementation

Before we go over to the actual pseudocode explaining the different sampling methods are implemented, we first discuss important notes for the framework.

4.1 PyTorch

The framework AL4PDE and therefore our extension is written with usage of PyTorch[1]. Pytorch is an open source machine learning library developed by Meta. It mainly resolves around the class Tensor, which is a multi-dimensional matrix storing and operating on a single data type. Both the ICs and PDE parameter are stored as such. If an operation is performed using a Tensor, it is stored in an internal graph. PyTorch also provides a method we can apply on a Tensor called "backward()". Here the graph is differentiated using the chain rule, and the gradient is therefore calculated for the current tensor, with regards to the graph leaves. We will utilize this method to calculate the logarithmic gradient of the uncertainty, which we require for the Langevin diffusion.

4.2 PDE and IC parameters

Since we are in the context of a Markov chain, we always have the data underlying from the previous state. Intuitivly, the data would be the PDE parameter and the IC from the previous state, but in the framework AL4PDE a small change is done. Instead of directly sampling from the PDE parameters, we sample from a normalized version, with a value between 0 and 1. To transform the normalized parameters to the correct ones, firstly the minimum and maximum value of the PDE are calculated:

$$cont_{min} = [\alpha_{min}, \beta_{min}, \gamma_{min}] + [A_{min}] \cdot 5 + [\omega_{min}] * 5 + [0] \cdot 5 \tag{4.1}$$

$$cont_{max} = [\alpha_{max}, \beta_{max}, \gamma_{max}] + [A_{max}] \cdot 5 + [\omega_{max}] * 5 + [2\pi] \cdot 5 \tag{4.2}$$

Each parameter α, β, γ has the minimal value 0 and the maximum value of 1. The same holds true for the amplitude A and the phase ω. Using the values of $cont_{min}$ and $cont_{max}$, the PDE parameter is calculated via:

$$cont = cont_{norm} \cdot (cont_{max} - cont_{min}) + cont_{min} \tag{4.3}$$

with $cont_{norm}$ being the normalized parameter. This behaviour is similar for the ICs. Firstly, only parameters for the ICs are fetched, and evaluated to the final ICs, with the PDE parameters. The IC parameters are defined through the underlying PDE, e.g. an amplitude or a phase.

[1] https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html

Algorithm 4.1 MALA proposal generation

```
function GENERATE_PROP(icp, pdepn, step_size)        // step_size learning rate
    icp_prop = N(icp + step_size/2 · icp.grad, step_size · I).sample()
    pdepn_prop = N(pdepn + step_size/2 · pdepn.grad, step_size · I).sample()
    return icp_prop, pdepn_prop
end function
```

Algorithm 4.2 Uncertainty gradient evaluation

```
function UNCERTAINTY_CALCULATION(ic, pdep)
    unc = model_uncertainty(ic, pdep)
    log_unc = log(unc)
    log_unc.backward()
end function
```

Since the maximum value of the amplitude and the phase is 1 and the minimal value is 0, a fetched value outside of the boundary is cropped back to the boundary value.

4.3 MALA

The first step to a working MALA implementation, is defining the proposal kernal described in Eq. 3.4. If we use that knownledge of the parameters for the construction of the Markov chain, instead of directly create a proposal for the ICs and PDE parameters, we rather create a proposal for the IC parameters and the normalized PDE parameters. This process is demonstrated in Alg. 4.1. Where icp represents the parameters for an initial condition, and $pdepn$ for the normalized PDE parameters. The $step\_size$ is the learning rate which we can set via a hyperparameter. I is the identity and we take a sample from the normal distribution using $.sample()$.

To get the values for $icp.grad$ and $pdep.grad$ we have access to the gradient of icp and $pdep$, which was calculated by applying the $.backward()$ from PyTorch function on the uncertainty. Crucially, if a propsal is outside of the definition boundary, it is cropped back to the definition boundary. This could potentially lead to duplicates, however with a small enough step size, this problem can be avoided. After we generate a proposal (and for the initial Markov state) we calculate the gradient on the logarithmic uncertainty shown in Alg. 4.2:

As soon as a propsal was made, we perform the Metropolis-Hastings step done for MALA. For the calculation we will use the transformation described in Sec. 3.5.1. If the ratio is higher than a random number between 0 and 1, we accept. By accepting, we save the proposed values for the IC and PDE parameters in the Markov chain, represented by a list, and use the values as base for our next iteration. If we reject, the old values are saved in the Markov chain and used again as base for our next iteration. This comes with the consequence of having samples multiple times in the Markov chain. The metropolis can be seen in Alg. 4.3 With $unc\_old(IC, \lambda)$ we denote the uncertainty created with the IC and the PDE parameters λ. If we put everything together, the whole code is in Alg. 4.4. To start the Markov chain, the first values are drawn at random, from samples that have not been trained on.

Algorithm 4.3 Metropolis-Hastings step

```
function ACCEPT_OR_REJECT(unc_old(IC_0, λ_0), unc_prop(IC_1, λ_1), complete_list)
    α = unc_old(IC_0, λ_0)/unc_prop(IC_1, λ_1) * q(X_t|X_{t+1})/q(X_{t+1}|X_{t+1})
    if random() <= α then                                              // accept
        complete_list.append(IC_params_1, IC_1, λ_norm_1, λ_1)
        return IC_1, IC_params_1, λ_1, λ_norm_1, unc_prop
    else                                                               // reject
        complete_list.append(IC_params_0, IC_0, λ_norm_0, λ_0)
        return IC_0, IC_params_0, λ_0, λ_norm_0, unc_old
    end if
end function
```

Algorithm 4.4 Pseudocode for MALA

```
function MALA(chain_size, step_size)
    complete_list = []
    ic, icp, p, pn = initialize_first_values()
    unc_cur = uncertainty_calculation(ic, p)
    for i in range(chain_size) do
        icp_prop, pdepn_prop = generate_proposal(icp, pn, step_size)
        pdep_prop = get_pde_params(pdepn_prop)
        ic_prop = get_ic(icp_prop, pdep_prop)
        unc_prop = evaluate_uncertaint(ic_prop, pdep_prop)
        ic, icp, p, pn, unc_cur = accept_or_reject(unc_cur, unc_prop, complete_list)
    end for
    return complete_list
end function
```

At the end we have a Markov chain containing *chain_size* samples. We then used the whole chain as input for our numerical solver to generate a solution and thereby used as training set for our neural PDE solver.

4.4 SC-ULA

For the implementation of SC-ULA, instead of the Metropolis-Hastings step, it rather just updates all parameters accordingly. The created chain then is further processed by the numerical solver.

4.5 PC-ULA

To modify SC-ULA for use in PC-ULA, we can use SC-ULA as a base. Instead of initializing a single tuple of values, we create *n* tuples in parallel. Since we're using PyTorch, parallel computation is supported by default, so the core code remains largely unchanged-only the dimensionality of the tensors increases.

Algorithm 4.5 Pseudocode for SC-ULA

```
function SC-ULA(chain_size, step_size)
    complete_list = []
    ic, icp, p, pn = initialize_first_values()
    unc_cur = uncertainty_calculation(ic, p)
    for i in range(chain_size) do
        complete_list.append(ic, icp, p, pn)
        icp_prop, pdepn_prop = generate_proposal(icp, pn, step_size)
        pdep_prop = get_pde_params(pdepn_prop)
        ic_prop = get_ic(icp_prop, pdep_prop)
        unc_prop = evaluate_uncertainty(ic_prop, pdep_prop)
        ic, icp, p, pn, unc_cur = ic_prop, icp_prop, pdep_prop, pdepn_prop, unc_prop
    end for
    return complete_list
end function
```

Algorithm 4.6 Pseudocode for PC-ULA

```
function SC-ULA(chain_size, step_size, n)
    complete_list = []
    ic, icp, p, pn = initialize_n_first_values(n)
    unc_cur = uncertainty_calculation(ic, p)
    for i in range(chain_size) do
        complete_list.append(ic, icp, p, pn)
        icp_prop, pdepn_prop = generate_proposal(icp, pn, step_size)
        pdep_prop = get_pde_params(pdepn_prop)
        ic_prop = get_ic(icp_prop, pdep_prop)
        unc_prop = evaluate_uncertainty(ic_prop, pdep_prop)
        ic, icp, p, pn, unc_cur = ic_prop, icp_prop, pdep_prop, pdepn_prop, unc_prop
    end for
    complete_list = complete_list(−n :)          // Take the last n-values
    return complete_list
end function
```

The algorithm begins by sampling the first n values, which serve as the starting points for the chains. It then calculates the uncertainty for each of these n values and generates the corresponding proposal states in parallel. At the end of the process, the algorithm discards the first $n - 1$ states, retaining only the final one.

5 Related Work

There are a number of different approaches and ideas for computing the solution trajectories for PDEs with the help of neural networks. The focus of this work is on neural PDE solvers that incorporate active learning strategies. Li et al. (2021) proposed an implementation in a multi-fidelity model to predict PDE solutions [LKZ21]. Active learning was used to learn a mapping for the PDE parameters from a low input dimension to a high output dimension. The acquisition function they chose was an extension of predictive entropy. This approach was further extended to work not only with single inputs, but also with batches. [LPY+22].

Multi-fidelity learning was then applied on a Fourier Neural Operator (FNO) to predict individual settings for the perfect spatial resolution [LYX+24].

Another baseline of underlying models are physics-informed neural networks (PINNs), which also include laws of physics for their loss function. Active learning in this context was proposed by Arthurs and King (2021). They proposed a method that integrates residual-based error metrics into the learning loop [AK21]. Their algorithm uses the residual error - the difference between the predicted and true values under the governing physical equations - as a guidance signal to iteratively refine the training data. The points with the largest residual error were sampled to the numerical solver - in this case based on the finite element method (FEM).

In the area of surrogate modelling, Pestourie et al. (2020) demonstrated the effectiveness of uncertainty-based active learning in the context of solving Maxwell's equations for optical metasurface design [PMN+20]. Their approach used an ensemble of neural networks to estimate prediction uncertainty. This method achieved surrogate model performance about two orders of magnitude faster than pure random sampling.

Extending the idea to another physical system, Bajracharya et al. (2024) investigated active learning strategies for DNN surrogate modelling of steady-state diffusion with sources [BTF+24]. They compared two uncertainty-based active learning strategies: entropy sampling and Temporal Output Discrepancy (TOD), which quantifies model uncertainty based on the difference in predictions between training simulations for the same input.

However, all of the work explained above are pool-based methods. For contrast, Dymchenko et al. (2024) proposed an active learning method for neural PDE solvers that does not rely on a fixed data pool, but instead performs on-the-fly query synthesis using a variant of Population Monte Carlo (PMC) sampling. After an initial iteration with uniformly sampled parameters, their method, called Breed, assigns each input an informativeness score based on the deviation of its training loss from the batch average. These scores are then used as importance weights to build a proposal distribution over the input space. New samples are drawn from this learned distribution, allowing the model to focus on harder-to-learn regions of the parameter space. While Breed showed improved generalisation and reduced overfitting in some configurations, its overall performance did not consistently exceed that of random sampling in the relatively simple heat PDE benchmark.

Finding further examples of query synthesis, even without the context of neural PDE solvers, is quite difficult. A fundamental requirement for query synthesis is the ability to generate new queries de novo. The challenge here is to ensure that the sample can be correctly labeled. If the underlying oracle is a human, the newly generated query may be unintelligible, as discussed by Bauma and Lang [BL92].

To make query synthesis practical, the choice of oracle must be carefully considered. A successful approach was presented by Wang et al. (2015) by combining a human oracle with a pool-based active learning approach. They generated samples close to the classification boundary in order to maximise the informativeness of the samples. However, since direct querying of synthetic samples was not feasible with a human oracle, they instead used a pool of real-world instances and selected the instance in the pool closest to the generated query. This query was then labeled and used for training.

The effectiveness of a human oracle also depends on how discriminative the sample categories are. In their work, Zhu and Bento (2017) modified an existing dataset by replacing images of cats and dogs with horses and cars, which are easier for humans to distinguish. As a measure of information, they also used proximity to the decision boundary as a criterion for informativeness. However, they found that their method performed similarly or slightly worse than traditional pool-based methods. One explanation they proposed is that query synthesis can theoretically produce an infinite number of instances close to the decision boundary - many of which are very similar to each other. In contrast, pool-based methods work with a finite set of samples, and with enough samples, the method is forced to query samples further away from the boundary, resulting in a more diverse sample.

However, these challenges do not apply to the present work. Because we use a numerical solver rather than a human, all samples are recognisable and interpretable. This is emphasised by the fact that we do not create queries completely de novo, but sample from an infinite input space.

6 Experiments

All experiments are based on the AL4PDE framework. For this work, we sample from the Combined Equation (CE), proposed by Brandstetter et al. (2022):

$$[\partial_t u + \partial_x(\alpha u^2 - \beta \partial_x u + \gamma \partial_{xx} u)] = 0, \tag{6.1}$$

It is called combined because the PDE parameters give rise to different equations. If we summarise the parameters as $\theta_{PDE} = (\alpha, \beta, \gamma)$, we have $\theta_{PDE} = (0, \eta, 0)$, which is the heat equation. Or, with $\theta_{PDE} = (0.5, \eta, 0)$ we get the Burgers equation [Bat15], which is used for fluid simulations. The boundary condition is periodic, and we have a slightly different IC generator that removes random frequency bins. This is required to allow us to get a continuous function for the gradient calculation. As PDE parameters for the input space we have α, β and γ. The experiments were conducted using an ensemble of two U-Nets. The query synthesis method is compared against random sampling, which queries random samples from the input space, and the old pool-based approach, with different pool sizes.

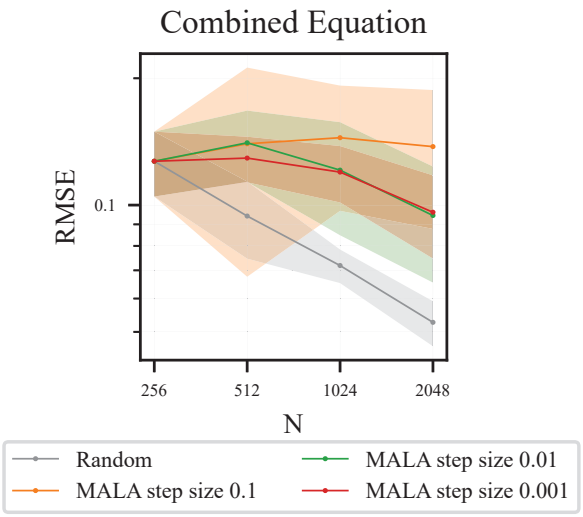
6.1 Metropolis adjusted Langevin algorithm

The error for the MALA algorithm with different step sizes is shown at Fig. 6.1. We can see on the x-axis the required amount of samples for our numerical solver to label. The required amount of samples is doubled each active learning iteration, therefore the chain size also has to be doubled, starting with 256 samples. On the y-axis we see the root mean square error (RMSE). Since we compare the loss to a random sampling, we can conclude MALA, is decreasing the loss slower than just random sampling. In this case, since even random sampling yields better results, there is no need to further compare it to the currently existing AL strategies. The main reason for the marginal decrease in loss is the repeated occurrence of samples in a chain. If a sample appears multiple times in the chain and is queried repeatedly - paired with its label - to the surrogate neural model, the model has already extracted the relevant information during the first query. As a result, the additional instances contribute little to no new information, so these duplicate samples are redundent. With a lower step size, MALA is improving, due to the higher acceptance rate.

6.2 SC-ULA

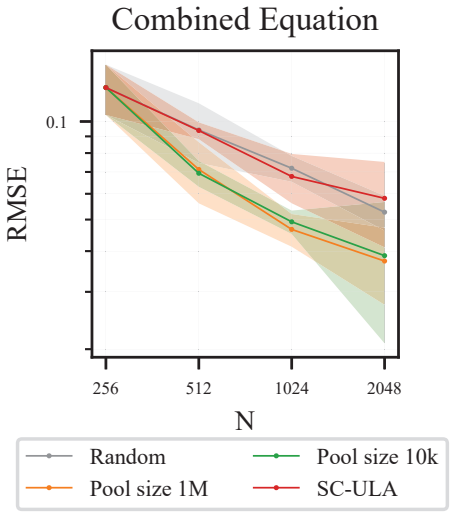
We solve this by removing the Metropolis-Hastings step, and just accepting every state proposed. This comes with the disadvantage of not converging to the target density π, but we will still sample from regions with higher uncertainties. By removing the acceptance step we are sampling with the unadjusted Langevin algorithm, described in Section 3.4. The error function, with the error on the

Figure 6.1: Loss functions from MALA with different chain sizes, and a uniformly random selection. The error from each MALA implementation is only decreasing slowly.



y-axis and the amount of samples on the x-axis, is displayed in Fig. 6.2. As we can see, the loss is steadily decreasing in contrast to the previous MALA implementation. Hoever, SC-ULA is still only comparable to a uniformly random selection.

Figure 6.2: Loss functions from SC-ULA, uniformly random selection and two SBAL implementations with different pool sizes. The error from the SC-ULA implementation is decreasing at the same rate as the random selection.

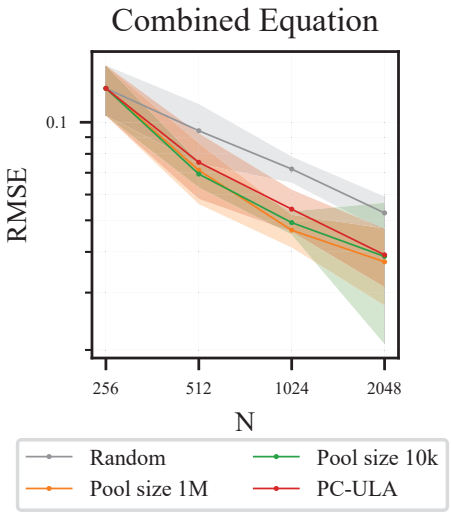


6.3 PC-ULA

Since we use the gradient to sample from higher uncertainties, and repeating this process over and over again, later states in the Markov chain should be more beneficial for us then earlier ones. We can make use of that by instead of using the complete chain as input, we rather only take later states. This corresponds to PC-ULA, described in Section 3.4.

To still match the amount of samples we use as input for the numerical solver, we also need more chains. If we only take the last state, the chain size can be choosen freely - for the experiments we set it to 1024 - but we require an amount of parallel chains matching the input for the numerical solver. The error function of PC-ULA, compared against a random selection and two SBAL methods, with the pool-size being ten thousand and one million are shown in Fig. 6.3 Again we can see the loss on

Figure 6.3: Loss functions from PC-ULA, uniformly random selection and two SBAL implementations with different pool sizes. The error from the PC-ULA implementation is decreasing at the same rate as both SBAL varaitions.



the y-axis and the numbers of samples on the x-axis. As we can see, even with parallel chains we are currently as good as the current implementations, but not better. If we look at the selection time in Fig. 6.4, we can see that the doubling of the samples goes hand in hand with a doubling of the selection time, which is given in seconds on the y-axis. This is due to the duplication of the amount of initial samples we use for the chain creation. We can reduce the chain size per sample to decrease our selection cost. This is illustrated in Fig. 6.5. For all chain sizes we use the same ULA algorithm with the same number of parallel samples. The only difference is the number of times ULA is applied to generate the next sample. We can see quite clearly that if we halve our chain size, the selection time is also halved. However, there is a downside. As the number of steps the algorithm does towards the gradient is reduced, the number of samples with higher uncertainties is also reduced. This is visualized in Fig. 6.6. With respect to the limited amount of simulations done, we can see that the longer the chain, the better the reduction of the loss.

Figure 6.4: The selection time of PC-ULA in comparison to uniformly random selection and two SBAL implementations with different pool sizes. The selection rate of PC-ULA is initially at the same value as SBAL with a pool size of one million, but is proportional to the number of samples queried.

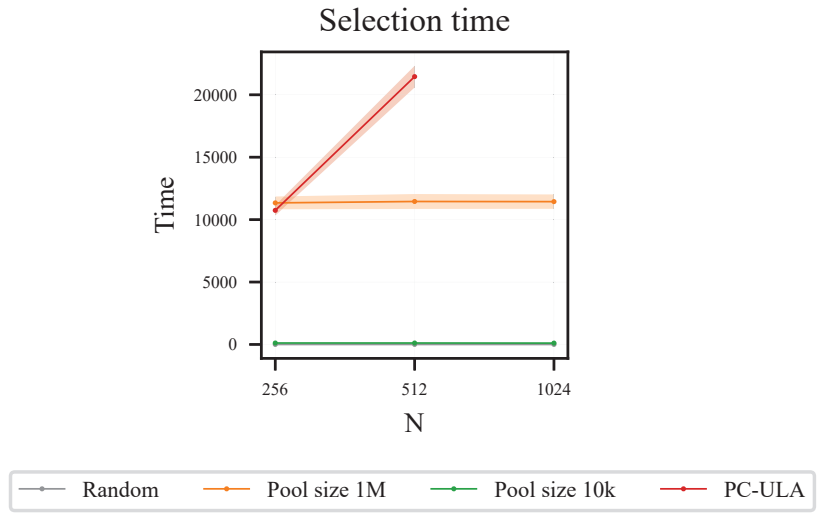
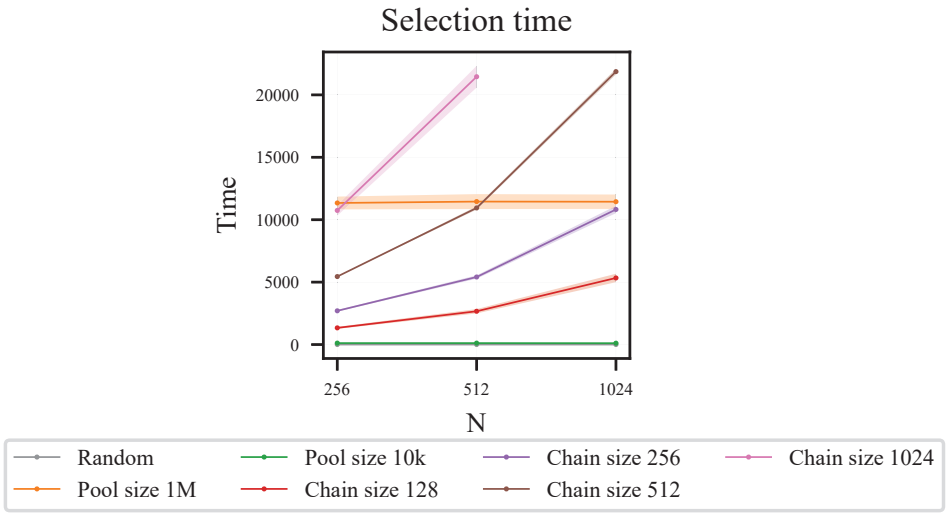
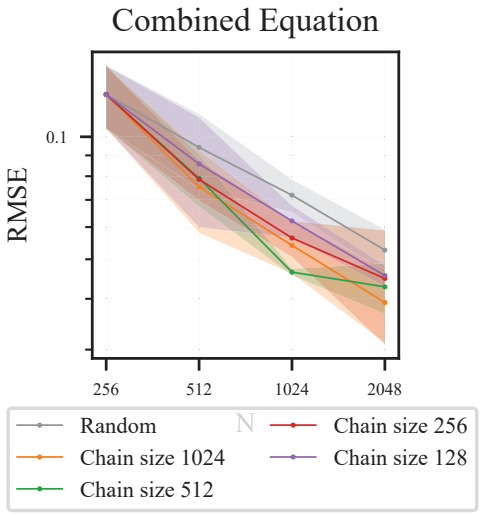


Figure 6.5: The selection time of PC-ULA with different chain sizes. This illustrates the proportionality between chain size and selection time.



To have a clearer understanding on how the parameters are changed, we can have a look at the changing rates from an example chain. In Fig. 6.7 we can see on the x-axis the 1024 states from the Markov chain, with the values on the y-axis. Through the small changes per sample, the dependency of the previous state is vividly illustrated.

Figure 6.6: The loss function is illustrated, with different chain sizes for PC-ULA in comparison to a random selection. Although each different chain size improved compared to the random selection, if one chain was longer than another, the error was reduced at a higher rate.



The initial conditions are split between the amplitude and the phase. To have a visually appealing graph, they are displayd seperatly. The amplitdude is displayed in Fig. 6.8 and the phase in Fig. 6.9. As we can see if the proposals are propsing new states further than the boundary, they are cropped back to the edges of the boundary conditions.

Figure 6.7: Changes in the PDE parameters per state for one chain with the size 1024. Because one state is dependent on the previous state, the changes between two neighbors are relatively small.

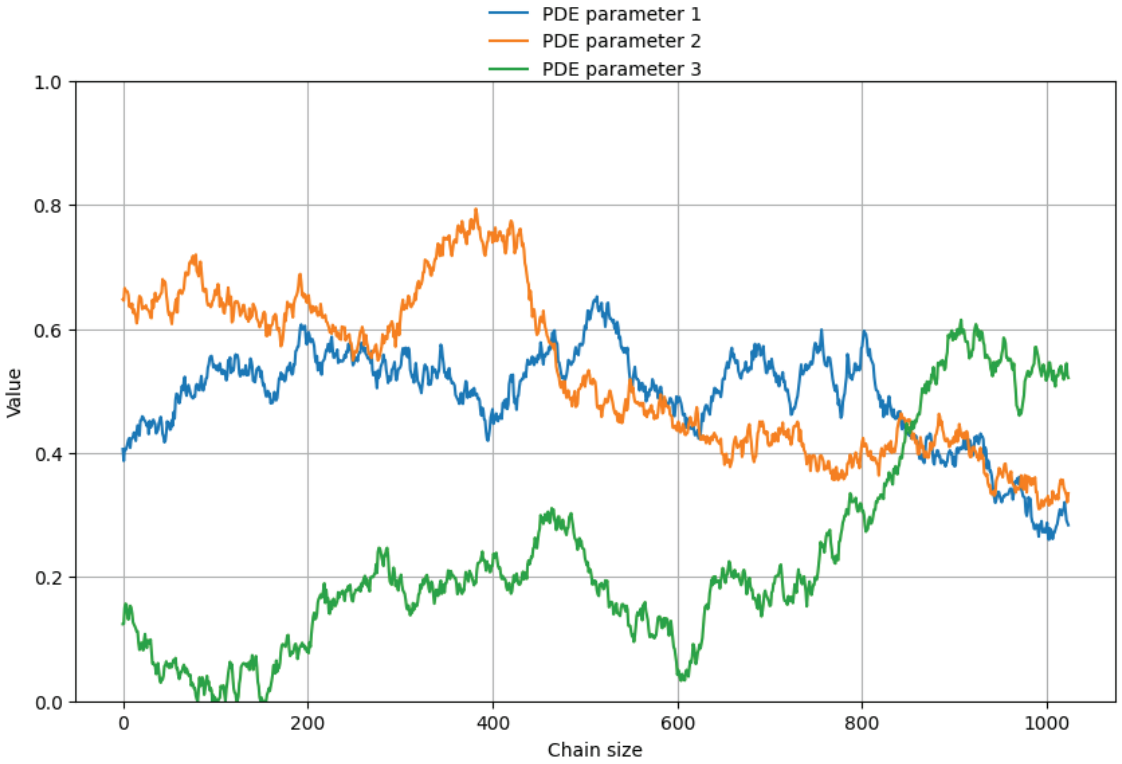


Figure 6.8: Changes in the amplitude for the IC per state for one chain with the chain size 1024. Values exceeding the definition boundaries are cropped, and because of the dependency, changes between neighbors are relatively small.

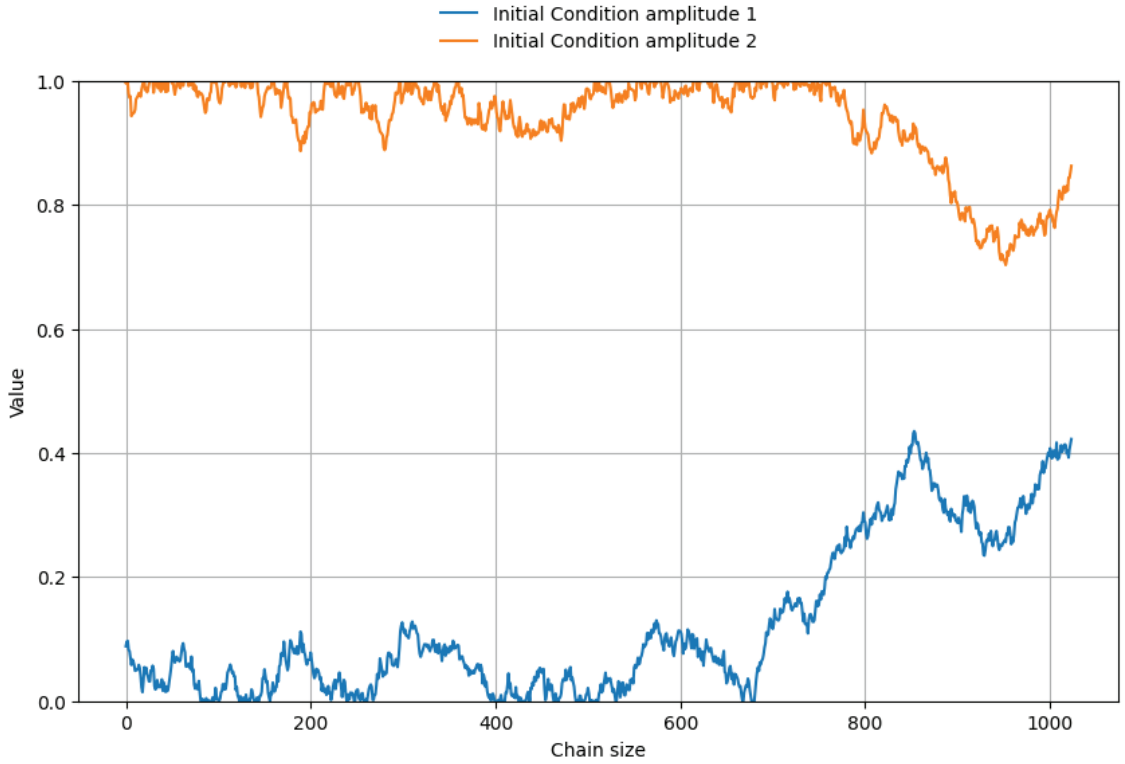
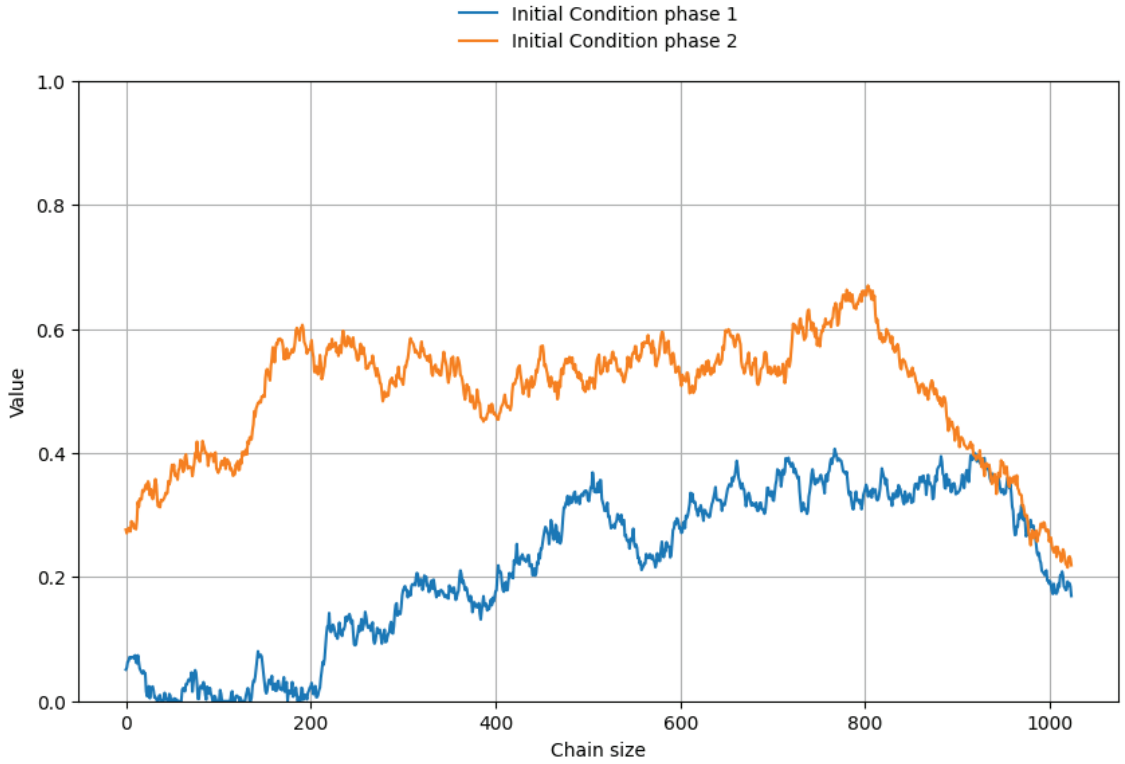


Figure 6.9: Changes in the amplitude for the IC per state for one chain with the chain size 1024. Values exceeding the definition boundaries are cropped, and because of the dependency, changes between neighbors are relatively small.



7 Conclusion and Outlook

The aim of this work was to implement a new active learning (AL) strategy for the existing framework AL4PDE. AL4PDE is an active learning framework in which an ensemble of neural networks approximates the solution trajectories of a PDE. The trainings data consits of a specific PDE with the associated parameters and initial conditions, paired with the belonging solution trajectory. The solution is provided by a numerical scheme, depending on the PDE. Our proposed method retains the core principle of using model uncertainty - derived from variance across the ensemble - as a measure of sample informativeness.

Previously, the existing AL strategy uses a pool-based sampling strategy, where a large set of samples is derived from the input space and the uncertainty is evaluated. However, this approach requires calculating the uncertainty for each sample in the pool, which is computationally expensive and limits the sampling to the given pool.

To address this, we proposed a query synthesis approach that samples directly from the input space, eliminating the need for pool creation and allowing for more flexible exploration. Crucially, in our case, the gradient of uncertainty is available, allowing the method to steer sampling towards globally more informative regions.

For query synthesis, we use Markov Chain Monte Carlo (MCMC) methods, where the target distribution is the uncertainty. Our initial strategy used the Metropolis-adjusted Langevin algorithm, which combines both a discretised version of Langevin diffusion and a Metropolis-Hastings step. Thus, it satisfies both the convergence to uncertainty and the gradient information. The chain was used as input to generate training data. However, the results showed that repeated evaluation of rejected samples led to inefficient learning, as the ensemble had already learned from these instances, resulting in limited loss reduction.

To mitigate this, we removed the Metropolis-Hastings step and accepted all proposed states unconditionally. While this improved performance over MALA, it was still inferior to the original pool-based method. To further improve efficiency, we introduced a parallelized approach where multiple chains were initialized with independent proposals and only the final state of each chain was used for training. The chain length was controlled by a hyperparameter.

This parallelized strategy produced results comparable to the original method. However, due to the computational cost of maintaining multiple chains - especially for longer chain lengths - our method became slower than pool-based sampling with large pool sizes. We showed that reducing the chain length per sample mitigated this cost, albeit at the cost of increased loss.

Based on the foundation laid in this work, future improvements should include sampling techniques utilizing an adaptation of MCMC while not being dependent on the Metropolis-Hastings acceptance step. One potential way of achieving this with the current implementation is to start MALA on already trained samples. Therefore, by proposing the next step, the probability of acceptance should be higher, since unknown samples typically have a higher uncertainty than samples already trainend

on. However, the problem with duplicates will probably still exist. One further usage could be to not use the chain as input. Instead, use it to create the pool in pool-based methods. This could lead to a pool with samples containing higher uncertainties, makeing future sampling more effective.

More broadly, advanced sampling methods, such as Hamiltonian Monte Carlo (HMC) and Population Monte Carlo (PMC) offer promising directions.

HMC uses Hamiltonian dynamics to propose transitions in the input space, using both gradients and momentum variables to generate proposals that follow the target distribtution more efficiently. HMC avoids random walk behaviour, allowing faster convergence - particular in high dimensional spaces like the PDE input space. HMC is also the more general form of MALA - or the other way around, MALA is a specific implementation of HMC. However, since HMC still has an acceptance step it could potentially run into the same errors as MALA.

If we want to completly get rid of the Metropolis-Hastings step, PMC would be an alternative. PMC offers a population-based approach where multiple samples are evolved simultaneously, with importance weights guiding resampling and adaptation over iterations. PMC also naturally supports parallelism and could scale well with our existing parallel chain architecture.

In summary, our work presents a viable alternative to pool-based sampling in AL4PDE, offering a gradient-informed and more flexible sampling methods. While still trade-offs in runtime and accuracy remain, our approach lays the groundwork for future improvements in efficient, uncertainty-guided query synthesis for PDE-based learning frameworks.

Bibliography

[AK21] C. J. Arthurs, A. P. King. "Active training of physics-informed neural networks to aggregate and interpolate parametric solutions to the Navier-Stokes equations". In: *Journal of Computational Physics* 438 (2021), p. 110364. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2021.110364. URL: https://www.sciencedirect.com/science/article/pii/S002199912100259X (cit. on p. 31).

[Ang88] D. Angluin. "Queries and Concept Learning". In: *Machine Learning* 2.4 (Apr. 1988), pp. 319–342. ISSN: 1573-0565. DOI: 10.1023/A:1022821128753. URL: https://doi.org/10.1023/A:1022821128753 (cit. on pp. 13, 19).

[Ayo10] T. O. Ayodele. "Machine learning overview". In: *New Advances in Machine Learning* 2.9-18 (2010), p. 16. DOI: 10.5772/9374 (cit. on p. 16).

[Bat15] H. Bateman. "Some recent researches on the motion of fluids". In: *Monthly Weather Review* 43.4 (1915), pp. 163–170. DOI: hhttps://doi.org/10.1175/1520-0493(1915)43<163:SRROTM>2.0.CO;2. URL: https://journals.ametsoc.org/view/journals/mwre/43/4/1520-0493_1915_43_163_srrotm_2_0_co_2.xml (cit. on p. 33).

[BL92] E. B. Baum, K. Lang. "Query learning can work poorly when a human oracle is used". In: *International joint conference on neural networks*. Vol. 8. Beijing China. 1992, p. 8 (cit. on p. 32).

[BTF+24] P. Bajracharya, J. Q. Toledo-Marín, G. Fox, S. Jha, L. Wang. *Feasibility Study on Active Learning of Smart Surrogates for Scientific Simulations*. 2024. arXiv: 2407.07674 [cs.LG]. URL: https://arxiv.org/abs/2407.07674 (cit. on p. 31).

[BWW22] J. Brandstetter, D. E. Worrall, M. Welling. "Message Passing Neural PDE Solvers". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=vSix3HPYKSU (cit. on p. 16).

[Gey11] C. J. Geyer. "Introduction to markov chain monte carlo". In: *Handbook of markov chain monte carlo* 20116022.45 (2011), p. 22 (cit. on p. 23).

[Has70] W. K. Hastings. "Monte Carlo sampling methods using Markov chains and their applications". In: *Biometrika* 57.1 (Apr. 1970), pp. 97–109. ISSN: 0006-3444. DOI: 10.1093/biomet/57.1.97. eprint: https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf. URL: https://doi.org/10.1093/biomet/57.1.97 (cit. on p. 24).

[KFA+21] A. Kirsch, S. Farquhar, P. Atighehchian, A. Jesson, F. Branchaud-Charron, Y. Gal. "Stochastic batch acquisition: A simple baseline for deep active learning". In: *arXiv preprint arXiv:2106.12059* (2021) (cit. on pp. 18, 19).

[LBH15] Y. LeCun, Y. Bengio, G. Hinton. "Deep learning". In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539 (cit. on p. 13).

[Lew95] D. D. Lewis. "A sequential algorithm for training text classifiers: Corrigendum and additional data". In: *Acm Sigir Forum*. Vol. 29. 2. ACM New York, NY, USA. 1995, pp. 13–19 (cit. on p. 13).

[LKZ21] S. Li, R. M. Kirby, S. Zhe. *Deep Multi-Fidelity Active Learning of High-dimensional Outputs*. 2021. arXiv: 2012.00901 [cs.LG]. URL: https://arxiv.org/abs/2012.00901 (cit. on p. 31).

[LPY+22] S. Li, J. M. Phillips, X. Yu, R. Kirby, S. Zhe. "Batch Multi-Fidelity Active Learning with Budget Constraints". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 995–1007. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/06ea400b9b7cfce6428ec27a371632eb-Paper-Conference.pdf (cit. on p. 31).

[LVP+23] P. Lippe, B. Veeling, P. Perdikaris, R. Turner, J. Brandstetter. "PDE-Refiner: Achieving Accurate Long Rollouts with Neural PDE Solvers". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, S. Levine. Vol. 36. Curran Associates, Inc., 2023, pp. 67398–67433. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/d529b943af3dba734f8a7d49efcb6d09-Paper-Conference.pdf (cit. on p. 13).

[LYX+24] S. Li, X. Yu, W. Xing, R. Kirby, A. Narayan, S. Zhe. "Multi-Resolution Active Learning of Fourier Neural Operators". In: *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*. Ed. by S. Dasgupta, S. Mandt, Y. Li. Vol. 238. Proceedings of Machine Learning Research. PMLR, Feb. 2024, pp. 2440–2448. URL: https://proceedings.mlr.press/v238/li24k.html (cit. on p. 31).

[MKH+24] D. Musekamp, M. Kalimuthu, D. Holzmüller, M. Takamoto, M. Niepert. "Active Learning for Neural PDE Solvers". In: *arXiv preprint arXiv:2408.01536* (2024) (cit. on pp. 13, 22).

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller. "Equation of State Calculations by Fast Computing Machines". In: *The Journal of Chemical Physics* 21.6 (June 1953), pp. 1087–1092. ISSN: 0021-9606. DOI: 10.1063/1.1699114. eprint: https://pubs.aip.org/aip/jcp/article-pdf/21/6/1087/18802390/1087_1_online.pdf. URL: https://doi.org/10.1063/1.1699114 (cit. on p. 24).

[Nea+11] R. M. Neal et al. "MCMC using Hamiltonian dynamics". In: *Handbook of markov chain monte carlo* 2.11 (2011), p. 2 (cit. on p. 25).

[PMN+20] R. Pestourie, Y. Mroueh, T. V. Nguyen, P. Das, S. G. Johnson. "Active learning of deep surrogates for PDEs: application to metasurface design". In: *npj Computational Materials* 6.1 (Oct. 2020), p. 164. ISSN: 2057-3960. DOI: 10.1038/s41524-020-00431-2. URL: https://doi.org/10.1038/s41524-020-00431-2 (cit. on p. 31).

[Rek19] D. Reker. "Practical considerations for active machine learning in drug discovery". In: *Drug Discovery Today: Technologies* 32-33 (2019). Artificial Intelligence, pp. 73–79. ISSN: 1740-6749. DOI: https://doi.org/10.1016/j.ddtec.2020.06.001. URL: https://www.sciencedirect.com/science/article/pii/S1740674920300019 (cit. on p. 13).

[Ros+11] J. S. Rosenthal et al. “Optimal proposal distributions and adaptive MCMC”. In: *Handbook of Markov Chain Monte Carlo* 4.10.1201 (2011) (cit. on p. 25).

[RR98] G. O. Roberts, J. S. Rosenthal. “Optimal scaling of discrete approximations to Langevin diffusions”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 60.1 (1998), pp. 255–268. DOI: https://doi.org/10.1111/1467-9868.00123. eprint: https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/1467-9868.00123. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9868.00123 (cit. on p. 25).

[RSKB17] M. E. Ramirez-Loaiza, M. Sharma, G. Kumar, M. Bilgic. “Active learning: an empirical study of common baselines”. In: *Data mining and knowledge discovery* 31 (2017), pp. 287–313 (cit. on p. 16).

[RXC+21] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, X. Wang. “A survey of deep active learning”. In: *ACM computing surveys (CSUR)* 54.9 (2021), pp. 1–40 (cit. on pp. 17, 18).

[Set09] B. Settles. “Active learning literature survey”. In: *University of Wisconsin-Madison Department of Computer Sciences, Tech. rep* (2009) (cit. on pp. 13, 16, 17).

[SS17] O. Sener, S. Savarese. “Active learning for convolutional neural networks: A core-set approach”. In: *arXiv preprint arXiv:1708.00489* (2017) (cit. on p. 18).

[WHYL15] L. Wang, X. Hu, B. Yuan, J. Lu. “Active learning via query synthesis and nearest neighbour search”. In: *Neurocomputing* 147 (2015), pp. 426–434 (cit. on p. 19).

[ZLLC21] X. Zhan, H. Liu, Q. Li, A. B. Chan. “A comparative survey: Benchmarking for pool-based active learning.” In: *IJCAI*. 2021, pp. 4679–4686 (cit. on p. 18).

All links were last followed on April 25, 2025.