Cheetah: Bridging the Gap Between Machine Learning and Particle Accelerator Physics with High-Speed, Differentiable Simulations*

Jan Kaiser,[1, †] Chenran Xu,[2, ‡] Annika Eichler,[1, 3] and Andrea Santamaria Garcia[2]

[1]*Deutsches Elektronen-Synchrotron DESY, Germany*
[2]*Karlsruhe Institute of Technology (KIT), Germany*
[3]*Hamburg University of Technology, 21073 Hamburg, Germany*
(Dated: 11 January 2024)

Machine learning has emerged as a powerful solution to the modern challenges in accelerator physics. However, the limited availability of beam time, the computational cost of simulations, and the high-dimensionality of optimisation problems pose significant challenges in generating the required data for training state-of-the-art machine learning models. In this work, we introduce *Cheetah*, a PyTorch-based high-speed differentiable linear-beam dynamics code. Cheetah enables the fast collection of large data sets by reducing computation times by multiple orders of magnitude and facilitates efficient gradient-based optimisation for accelerator tuning and system identification. This positions Cheetah as a user-friendly, readily extensible tool that integrates seamlessly with widely adopted machine learning tools. We showcase the utility of Cheetah through five examples, including reinforcement learning training, gradient-based beamline tuning, gradient-based system identification, physics-informed Bayesian optimisation priors, and modular neural network surrogate modelling of space charge effects. The use of such a high-speed differentiable simulation code will simplify the development of machine learning-based methods for particle accelerators and fast-track their integration into everyday operations of accelerator facilities.

I. INTRODUCTION

Future particle accelerator experiments will place ever-increasing demands on the performance and capabilities of particle accelerator operations and experiment analysis. In order to meet these demands, the research community is increasingly turning to machine learning (ML) methods, which have already demonstrated their ability to push the envelope of what is possible in the field of accelerator science [1–4].

One of the remaining challenges holding back this line of research is the demand for large amounts of data (including environment interactions) of these methods. Reinforcement learning (RL), for example, has already successfully been used to train intelligent tuning algorithms and controllers that can outperform the currently deployed black-box optimisation algorithms and hand-crafted controllers [1, 5–7]. However, RL methods require many interactions with their target task to train a well-performing policy. For example, 6 000 000 samples were needed in [1] to successfully train a policy on a transverse beam tuning task. Orders of magnitude larger number of samples are also common with other RL applications [8, 9]. The general scarcity of beam time makes collecting experimental data for ML methods, such as RL, a significant bottleneck. Gathering at least partial data sets in simulation can alleviate this problem, but existing accelerator simulation codes have mostly been developed with a focus on the design phase of accelerators, where high-fidelity and physical correctness are critical, and computing times range from minutes to several hours for one simulation. Consequently, data collection with existing simulation codes becomes impractical with the growing demand for large data sets. In this paper, we introduce *Cheetah*, a PyTorch-based high-speed differentiable linear-beam dynamics code. Cheetah is capable of accelerating beam dynamics simulations by multiple orders of magnitude through tensorised computation and several speed optimisation methods. In the specific example of [1], this equates to a reduction in RL training time from over 12 days when using the Ocelot simulation code [10] to just over 1 hour when using Cheetah.

At the same time, numerical optimisation is fast becoming an important tool for accelerator design, tuning, and model calibration [11, 12]. Advanced numerical optimisation methods like Bayesian optimisation (BO) have been used to achieve impressive results [13]. However, demands to solve optimisation problems of increasing dimensionality are growing, and BO may struggle to efficiently optimise objective functions with more than a few dozen degrees of freedom [13]. In the field of ML, gradient-based optimisation has successfully been used to optimise up to 70 billion parameters [14, 15]. However, computing gradients of complex models, like beam dynamics, using numerical or analytical methods are computationally expensive. Instead, automatic differentiation has found widespread adoption in machine learning for the fast computation of gradients. ML frameworks, such as *PyTorch* [16] and *JAX* [17], allow convenient and computationally cheap automatic differentiation to calculate the partial derivatives up to arbitrary orders for all the parameters using the chain rule. Because Cheetah is constructed upon PyTorch, it provides built-in support

† jan.kaiser@desy.de; Equal contributions
‡ chenran.xu@kit.edu; Equal contributions

for automatic differentiation to efficiently compute the gradients of the beam dynamics models it implements. Hence, Cheetah makes optimisation over the large parameter spaces of accelerator facilities tractable beyond the number of parameters that can feasibly be optimised with the current state-of-the-art numerical optimisers.

Surrogate modelling of start-to-end accelerator systems utilising neural networks (NNs) is another active area of research [2–4]. Such surrogate models can be used to acquire offline models of processes in accelerator facilities or as a fast and differentiable stand-in for computationally expensive simulations. Nevertheless, NN are usually trained on start-to-end data, taking actuators as inputs and sensor values as output. This makes it difficult to reuse trained models for applications beyond those intended at the time of training. Moreover, NN surrogate models are not commonly designed to interface with beam dynamics simulators. Conveniently, Cheetah is implemented using PyTorch, which is first and foremost an ML framework. As a result, models implemented in Cheetah can be readily integrated with NN surrogate models. This also means that gradient propagation from NN surrogate models through Cheetah and vice versa works out-of-the-box. With Cheetah, it is therefore possible to combine modular NN surrogate models with physical beam dynamics simulations. In particular, Cheetah provides a practical platform for integrating modular NN surrogate models with handcrafted beam dynamics models, making the expensive-to-train surrogate models more reusable.

In the following, we introduce Cheetah and its inner workings in Section II, benchmarking its speed in Section II D. In the second half of this paper, we present five different application examples (also shown in context in Fig. 1), taking advantage of Cheetah's speed for reinforcement learning in Section III A, and using its differentiability for beam tuning in Section III B and system identification in Section III C, followed by an example using Cheetah as a BO prior in Section III D, and demonstrating how Cheetah may host modular NN surrogate models in Section III E.

A. Related work

The field of programmatic beam dynamics modelling is very mature. There exist various well-established simulation codes for modelling beam dynamics in particle accelerators, e.g. ASTRA [18], Bmad [19], Elegant [20], and MAD-X [21]. As Python has become increasingly popular in scientific computing, many of these have been augmented with Python adaptors. Further, the Ocelot [10], Xsuite [22], and Bmad-X [23] simulation codes have been specifically developed directly in Python. Some calculations in Cheetah are based on those used in Ocelot.

Neural network (NN) surrogate modelling is also finding increased use to acquire fast and accurate models of complex beam behaviours [2, 4]. An NN surrogate

model is trained to infer the space charge field in a vacuum chamber cross-section using a physics-informed loss function including a partial differential equation with the Lorentz factor, elliptical bi-Gaussian charge density, and boundary condition in [3].

An overview of opportunities for differentiable programming in particle physics instruments is given in [24].

Specialised handcrafted differentiable simulations have been constructed for various applications. A handcrafted differentiable physics model is used as the discriminator in a generative adversarial network (GAN) setup to train an NN to reconstruct time-domain measurements of X-ray pulses without labelled data in [25]. In [26, 27] the hysteron density function of a Preisach model is fitted to accurately model hysteresis from experimental data. In [27, 28], a differentiable beam dynamics simulation of a tomographic beamline is used to reconstruct phase space distributions from experimental screen images. Simultaneous calibration of all detector parameters of a liquid argon time projection chamber using a differentiable simulation of the latter is performed in [29]. In [30], a differentiable self-consistent space charge simulation model based on the truncated power series algebra (TPSA) is developed to speed up the simulated optimisation of accelerator design parameters under consideration of space charge induced effects.

A similar effort to Cheetah is pursued in [23], where the authors introduce *Bmad-X*, a library-agnostic differentiable particle tracking code written in Python based on Bmad. They demonstrate the application of Bmad-X on examples of beamline optimisation, model calibration, and phase-space reconstruction. Bmad-X and Cheetah present very similar advantages, with both offering fast differentiable beam dynamics simulations. However, they differ in some aspects. In contrast to Cheetah, Bmad-X can be used with backend libraries other than PyTorch, while Cheetah has a stronger focus on fast computations and currently supports a larger number of lattice elements and conversions from other simulation codes. Specifically, the goal of Cheetah is to bridge the gap between fast hand-crafted and data-driven particle accelerator models, streamlining their applications to various applications. As such, it aims to enable researchers to collect low-fidelity data fast and to use differentiable models to train ML models or perform complex system identification.

An early preliminary version of Cheetah was first presented in [31]. It was not yet designed to support automatic differentiation and lacked the majority of the features Cheetah now has.

II. FAST DIFFERENTIABLE LINEAR BEAM DYNAMICS IN PYTORCH

The overarching goal in implementing Cheetah was to provide a differentiable beam dynamics code with improved speed over existing simulation codes to be used
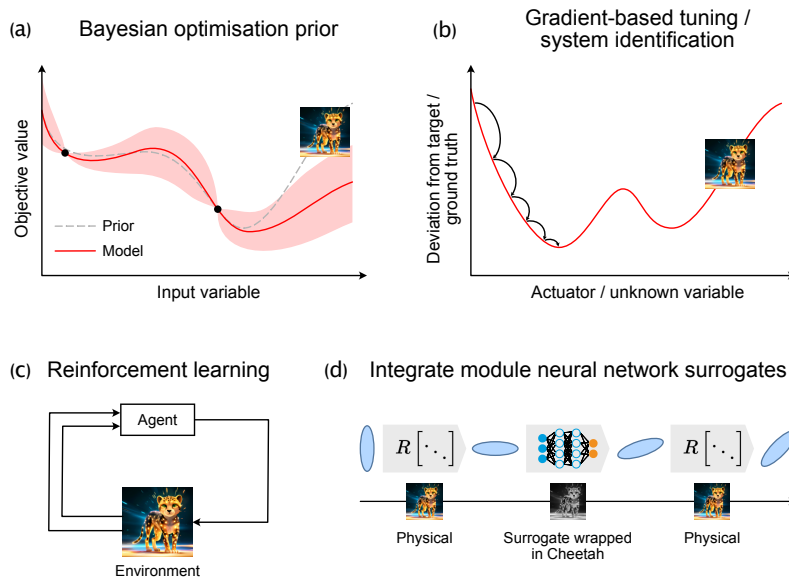
FIG. 1. Overview of where Cheetah fits into the proposed applications, with the Cheetah logo marking its use as a component of these applications. (a) Cheetah is used as a physical prior for BO. (b) Cheetah provides a differentiable beam dynamics model which can be used for accelerator tuning and system identification. (c) Cheetah enables the implementation of fast beam dynamics environments for training RL agents. (d) Cheetah provides the infrastructure to seamlessly integrate modular NN surrogate models with physical beam dynamics simulations.

for ML applications. Here, the conscious decision is made to trade accuracy to achieve these speed improvements. This means that Cheetah, while faster than existing codes, is lower fidelity than they are. With ML applications this is a worthwhile trade-off. Methods like domain randomisation [32] enable NN models trained on inaccurate simulated data to effectively generalise to the real-world domain. Moreover, initial training of an ML model on cheap low-fidelity data followed by fine-tuning on high-fidelity data is a widely used method to speed up the training of ML models. At the same time, Cheetah is designed to integrate seamlessly with popular ML tools. We intend for Cheetah to be used both as a tool in ML applications, e.g. to support the training of neural network models, and as an application of ML itself, e.g. through the integration of neural network models in its simulation pipeline. Last but not least, our goal is to make Cheetah easy to use, easily extensible and follow best practices in its implementation with high-quality code.

To this end, Cheetah is implemented in the *Python* programming language, which hosts an extensive ML ecosystem and is widely used in scientific computing. Cheetah employs the *PyTorch* [16] framework. While the primary purpose of PyTorch is the implementation of ML algorithms, its fast tensor compute capabilities, strong graphics processing unit (GPU) support and automatic differentiation features make it an ideal fit for fast parallel scientific computation.

To validate that Cheetah's models the physics of beam dynamics accurately and to ensure high code quality, Cheetah makes use of various continuous integration (CI) pipelines. Numerous tests are implemented to verify not only that Cheetah runs without errors, but also that its outcomes are physically plausible and match those computed by Ocelot [10, 11]. Automatic code formatting and linting are also used to enforce a high standard of readability and maintainability for Cheetah's code, while ensuring that implementations follow the best programming practices of PEP8 and minimising the incidence of elusive future errors. The official GitHub repository [33] is set up with clear contribution guidelines and well maintained in an effort to foster future collaboration in the development of Cheetah. To lower the barrier of entry and ease installation, stable versions of Cheetah are regularly deployed to *PyPI*. Reference documentation and some use case examples for Cheetah are made available via *Read the Docs*[34].

A. Beam tracking in Cheetah

At its core, Cheetah is made up of two main object classes, `Beam` and `Element`, which provide implementations of charged particle beams and accelerator elements, such as magnets and drift sections, respectively. Both of these inherit from PyTorch's `Module`, allowing their parameters to be optimised with the tools provided by PyTorch when set to a `Parameter` instead of a `Tensor`.

Cheetah provides two ways to represent the beam, a `ParticleBeam` with coordinates of each macroparticle and a `ParameterBeam` with only statistical values representing the beam, both being a subclass of the `Beam`.

In `ParticleBeam`, each particle is represented by a seven-dimensional vector

$$\boldsymbol{p} = (x, x', y, y', \tau, \delta, 1),\qquad(1)$$

where $\{x, y\}$ are the horizontal and vertical positions, $\{x', y'\}$ are the slopes in trace space, τ the longitudinal displacement, and δ the momentum offset with respect to the nominal energy. The six-dimensional vector is expanded at the end, analogous to an affine space, allowing a coherent representation of transfer maps also for effects like magnet misalignments and thin-lens magnets.

For applications that require faster computations and do not require modelling of the bunch substructures, a second representation, the `ParameterBeam`, is used. It assumes a Gaussian beam and represents the entire beam by a seven-dimensional vector μ of the mean position in each dimension of the phase space and a covariance matrix Σ.

Furthermore, the `Beam` subclasses implemented in Cheetah offer convenient computation of various of their properties. Both beam representations support generating Gaussian beams or being loaded from files saved by other particle tracking codes, a feature which is further discussed in Section II C. In addition, `ParticleBeam` instances can be generated with regularly spaced macroparticles.

The `Element` class represents accelerator beamline elements, such as magnets, drift sections, or diagnostic instruments. Each subclass must implement a `track` method that transforms an incoming beam to an outgoing beam that was affected by the element. To add a new element to Cheetah, one simply inherits from `Element` and implements the `track` method. The `track` method can implement arbitrary computations from simple matrix multiplications for first-order tracking to more complex behaviours like higher-order transfer maps for non-linear elements, beam image computations for diagnostics screens, or neural network inference. By default, Cheetah elements compute linear beam dynamics using an implementation of the linear transfer map $R_{\text{Cheetah}} \in \mathbb{R}^{7\times7}$ that is already provided

$$R_{\text{Cheetah}} = \left(\begin{array}{c|c} R_0 & \vdots \\ \hline 0\cdots0 & 1 \end{array}\right),\qquad(2)$$

with $R_0 \in \mathbb{R}^{6\times6}$ being the standard transfer matrix based on [35]. For some elements more complex behaviours are already implemented in Cheetah. For example, the transverse motion in accelerating cavities is modelled according to [36]. For the remainder of this paper, all transfer matrices R are assumed to be of the form R_{Cheetah}.

To track a beam through an element with a transfer matrix R, the default implementation either computes

$$P_{\text{out}} = P_{\text{in}}R^{\mathsf{T}}\qquad(3)$$

for a `ParticleBeam` $P_{\text{in}} \in \mathbb{R}^{n\times7}$ with n macroparticles, or

$$\begin{aligned} \mu_{\text{out}} &= R\mu_{\text{in}} \\ \Sigma_{\text{out}} &= R\Sigma_{\text{in}}R^{\mathsf{T}} \end{aligned}\qquad(4)$$

for a `ParameterBeam` with the characteristic parameters $\{\mu_{\text{in}}, \Sigma_{\text{in}}\}$.

For elements that only implement linear beam dynamics, it is therefore sufficient to implement a `transfer_map` method returning a first order transfer matrix R for the element. At the time of writing this paper, Cheetah has support for drift sections, dipole magnets with adjustable face angles (e.g. SBends and RBends), thin-lens corrector magnets, quadrupole magnets, cavities, beam position monitors (BPMs), markers, diagnostic screen stations, apertures, solenoid magnets, and elements with custom transfer maps. In addition, Cheetah provides a special `Segment` subclass of `Element`. It represents a sequential lattice of accelerator components and supports nesting of other smaller `Segment` elements.

We continue to extend Cheetah with new elements and features. In a further community-driven effort, users of Cheetah can add new features, such as elements, physical processes, and specialised transfer maps, according to task specific requirements.

B. Speed optimisation

Cheetah achieves its speed through several automatic and opt-in optimisations. First of these is the use of PyTorch, which is itself implemented in $C++$ and Compute Unified Device Architecture (CUDA) and is well-optimised thanks to widespread community support. PyTorch holds a key speed advantage over established packages like NumPy in its built-in ability to run on GPUs supporting CUDA or Metal Performance Shaders (MPS), which can provide significant speed improvements for massively parallel computations such as single particle tracking.

Moreover, Cheetah automatically identifies sequences of elements that can have their transfer matrices combined. We refer to this optimisation as *dynamic transfer map reduction*. For example, if a `Segment` is made up of an alternating sequence of dipole magnets and drift sections following linear beam dynamics, followed by an active diagnostic screen station and another sequence of alternating dipole magnets and drift sections, Cheetah will automatically recognise that the transfer matrices $\{R_{M1}, R_{D1}, R_{M2}, R_{D2} \ldots\}$ of the elements upstream of the screen station can be combined into a single transfer matrix $R_{\text{upstream screen}}$, and that the same can be done for the sequence of elements downstream of the screen station. A simple example following this description is shown in Fig. 2. This optimisation can be influenced by the user to some extent. Some elements, such as diagnostic screen stations and BPMs, support being activated
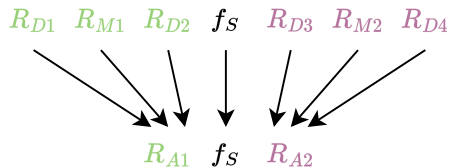
$$R_{D1} \quad R_{M1} \quad R_{D2} \quad f_S \quad R_{D3} \quad R_{M2} \quad R_{D4}$$

$$R_{A1} \quad f_S \quad R_{A2}$$

FIG. 2. Visualisation of a simple example for transfer map reduction. The tracking function of the screen is denoted by f_S. It cannot be reduced along with other transfer maps. The transfer maps drift sections and magnets upstream of the screen $\{R_{D1}, R_{M1}, R_{D2}\}$ and downstream of the screen $\{R_{D3}, R_{M2}, R_{D4}\}$ can be reduced to two transfer maps R_{A1} and R_{A2}, one on each end of the screen.

or deactivated, based on whether the user intends to use their functions. Cheetah makes transfer map reduction decisions based on the activation status of these elements. Other elements such as cavities, automatically determine whether they can be optimised through transfer map reduction. Cavities, for example, produce a drift transfer matrix when inactive, which is automatically combined with other transfer maps, but do not take part in transfer map reduction when they are active and have more complex effects on the beam. Because transfer map reduction does not always need to be performed every time a beam is tracked through a `Segment`, Cheetah provides an opt-in variant of the same optimisation, where the user can tell Cheetah which elements may be changed in the future. All other elements are then frozen, allowing Cheetah to perform *static transfer map reduction*. This optimisation can be very effective when only a few parameters are changed between consecutive simulations on large lattices.

In a similar vein, Cheetah can be commanded to find all inactive elements that are effectively drift sections, and replace them with actual drift sections, which are generally faster to compute. In addition, drift sections in Cheetah are pure linear beam dynamics elements, meaning they can be merged with adjacent linear beam dynamics elements, either dynamically and on-the-fly, or statically before tracking.

Especially when combining these optimisations, they can significantly speed up computations in Cheetah. Critically, all the implemented optimisations maintain the differentiability and correct gradients of the models.

C. Integration with other codes

To facilitate the quick adoption of Cheetah, the ability to load beams and lattices, especially from other particle tracking codes, is crucial. Cheetah's default lattice exchange format is an adapted variant of the interoperable lattice exchange format *LatticeJSON* [37]. Based on the standard JSON format, this makes reading and writing of lattice files which are compatible with Cheetah straightforward in any programming language. Cheetah's modular and simple architecture further simplifies the implementation of converters from other lattice and beam exchange formats. Cheetah currently supports loading beams from ASTRA, lattices from Bmad, and both beams and lattices from Ocelot.

D. Speed benchmarks

In this section, we benchmark the execution speed of Cheetah with other simulation codes on the same tracking tasks. The lattice considered for the benchmark is the *Experimental Area* section of the *ARES* accelerator [38, 39] at DESY, which is further investigated in some of the use case examples in Section III. The section is in total $2.05\,\mathrm{m}$ long, consisting of three quadrupole magnets, two corrector magnets, and drift sections in between. The benchmarks were run on two different computing platforms to account for the potential advantages of different hardware. Firstly, we ran simulations on a laptop with an Apple M1 Pro with 10 CPU Cores and $32\,\mathrm{GB}$ of RAM. Secondly, we considered a high-performance computing (HPC) cluster node with two AMD EPYC 7643 having a combined 192 cores, $1024\,\mathrm{GB}$ of RAM, and 4 Nvidia A100 GPUs, each having $80\,\mathrm{GB}$ of VRAM. Both the central processing unit (CPU) and the GPU were considered with Cheetah on the cluster node. Note that at the time of writing, Cheetah can only use one GPU at any time. Simulation times were averaged over multiple runs using Python's *timeit* package. Cheetah was run in multiple different configurations: tracking a `ParameterBeam`, tracking a `ParticleBeam` on CPU, and tracking a `ParticleBeam` on GPU. For all the configurations, we benchmarked with and without the opt-in lattice optimisations. We further compared Ocelot and ASTRA [18] with and without space charge. Parallel ASTRA was run using 8 performance cores on M1 Pro and 48 cores on EPYC 7643, which we found to be the fastest configurations for this particular benchmark. In addition, we consider Bmad-X with a NumPy backend and Xsuite for the benchmarks. The results of the speed benchmark are listed in Table I. For Cheetah with `ParticleBeam` and other simulation codes, a beam with $100\,000$ macroparticles is used. The benchmarks were run with a pre-release version of Cheetah v0.6.2.

We find that Cheetah can compute the benchmarked simulation setup up to 8 orders of magnitude faster than the other benchmarked simulation codes. In particular, Cheetah is about 5500 times faster than the fastest ASTRA setup without space charge on the ARM laptop. The fastest setup of Ocelot is outperformed by Cheetah by over 9000 times on the same device. In our benchmarks, Cheetah also achieves computational speeds around 1900 times faster than the already very fast Bmad-X. Xsuite achieves speeds comparable to Cheetah without Cheetah's opt-in optimisations turned on, but Cheetah is up to two orders of magnitude faster when

opt-in optimisations are used. Remember that these speed advantages of Cheetah by design come at the cost of accuracy, where higher-order effects, collective effects, and others are left out by default in order to achieve the reported speeds. We further find that in our benchmark Cheetah's `ParameterBeam` is tracked between 2 and 40 times faster than the same `ParticleBeam`. GPU acceleration is a sensible choice only with `ParticleBeam`, though it is not guaranteed to improve compute times. While we did observe 8 times faster simulation with lattice optimisations turned on, simulations slowed down by a factor of almost 1.6 without them. This is the result of the benchmark beam tracking 100 000 particles. In this case, the overhead induced by sending instructions and data to the GPU outweighs the performance benefits of highly-parallel computation. On the other hand, when the number of tracked particles is increased to 10 000 000, tracking with optimisations turned on takes 37.5 ms on CPU and 998 μs on GPU. With optimisations turned off, Cheetah tracks the same beam in 37.5 ms on CPU and 5.36 ms on GPU. This is a significant improvement, demonstrating the advantages of GPU acceleration in Cheetah. Moreover, we find that in our benchmarks, opt-in optimisations yielded up to 38 times faster execution on CPU and up to 51 times faster execution on GPU. Note that the opt-in optimisations benchmarked here are the most extreme case, taking maximum advantage of the optimisations to demonstrate Cheetah at its fastest and at its slowest. In real-world use of opt-in optimisations, we expect results to be slightly worse than the optimised cases showcased here, as some user-defined exceptions might reduce the effectiveness of Cheetah's optimisations.

III. USE CASE EXAMPLES

In the following we would like to demonstrate in five examples how Cheetah might be used and what it is capable of. In Section III A, we demonstrate on the example of recently published work, how Cheetah can be used to enable fast reinforcement learning in simulation for policies that transfer well to the real world. This is followed by examples of using Cheetah's automatic differentiation features to perform gradient-based accelerator tuning in Section III B and gradient-based system identification in Section III C. In Section III D, we show Cheetah's utility as a physics-based prior in the context of Bayesian optimisation. At last, we demonstrate Cheetah's suitability for an extension by modular element neural network surrogate models in Section III E. The following is not an exhaustive list of applications for Cheetah. We believe that as Cheetah is adopted, users will find many more problems it can solve.

A. High-speed simulations for reinforcement learning

In recent work, Cheetah played a key role in the successful training of a neural network policy for tuning the transverse beam properties in a particle accelerator through the method of RL [1, 5, 40] – so-called reinforcement learning-trained optimisation (RLO). Specifically, this work considers a tuning task in the Experimental Area (EA) beamline section at the *ARES* accelerator. The EA is made up of a sequence $\{Q_1, Q_2, C_v, Q_3, C_h\}$ of two quadrupole magnets, followed by a vertical dipole magnet, a third quadrupole magnet and a horizontal dipole magnet. These magnets allow for the tuning of the transverse beam properties $(\mu_x, \sigma_x, \mu_y, \sigma_y)$, i.e. position and size in the horizontal and vertical dimensions. These properties are measured with a diagnostic screen station downstream of the magnets. At ARES, transverse beam parameter tuning is commonly performed in preparations for experiments in an experimental vacuum chamber installed downstream of the EA. The goal of the transverse beam tuning task in the EA is to find the magnet settings $\boldsymbol{u} = (k_{Q_1}, k_{Q_2}, \alpha_{C_v}, k_{Q_3}, \alpha_{C_h})$ that minimise the difference between the beam parameters observed on the screen $\boldsymbol{b} = (\mu_x, \sigma_x, \mu_y, \sigma_y)$ and some target beam parameters $\boldsymbol{b}' = (\mu_x', \sigma_x', \mu_y', \sigma_y')$ set by a human operator. In the EA, the beam entering that section $\boldsymbol{b}_{in}$ and the transverse misalignments $\boldsymbol{m}$ of components like quadrupoles and the screen are not known, making this the transverse beam tuning task more challenging to solve. To date, transverse beam tuning is mostly solved manually by experienced human operators, which requires a lot of time and makes it difficult to reproduce results.

In order to solve this beam tuning task utilising RLO, a task-specific RL loop is defined as shown in Fig. 3. In this loop, the accelerator environment is implemented using Cheetah at the time of training and then replaced with the real ARES accelerator at the time of application. The Python package *Gymnasium* [41] (the successor to the previously popular *OpenAI Gym* [42]) is used to define the environment. A multilayer perceptron (MLP) of two hidden layers is used as a policy model. Each layer has a width of 64 neurons and uses a rectified linear unit (ReLU) activation. The policy takes as input the normalised observed beam $\boldsymbol{b}$, the currently set quadrupole strengths and deflection angles of the magnets $\boldsymbol{u}$, and the target beam parameters $\boldsymbol{b}'$. Its output is defined as normalised changes to the magnet settings $\boldsymbol{a}_t = \Delta \boldsymbol{u}$. The rewards and observations are normalised using a running average during the training. The actions are normalised to the action spaces, which is [-3, 3] m^{-2} for quadrupole strengths, [-0.6, 0.6] mrad for vertical steering magnet, and [-0.3, 0.3] mrad for horizontal steering magnet. The different action ranges of vertical and horizontal magnets are chosen so that they have approximately the same steering effect at the position of the diagnostics screen.

To train the policy, the Twin Delayed DDPG

TABLE I. Step computation times of simulation codes in milliseconds

| Code	Comment	Laptop	HPC node
ASTRA	space charge	264 000.00	3 605 000.00
	no space charge	109 000.00	183 000.00
Parallel ASTRA	space charge	39 000.00	17 300.00
	no space charge	16 900.00	12 600.00
Ocelot	space charge	22 100.00	21 700.00
	no space charge	182.00	119.00
Bmad-X		40.50	74.30
Xsuite	CPU	0.81	2.82
	GPU	-	0.57
Cheetah	`ParticleBeam`	1.60	2.95
	`ParticleBeam` + optimisation	0.79	0.72
	`ParticleBeam` + GPU	-	4.63
	`ParticleBeam` + optimisation + GPU	-	0.09
	`ParameterBeam`	0.76	1.29
	`ParameterBeam` + optimisation	0.02	0.04
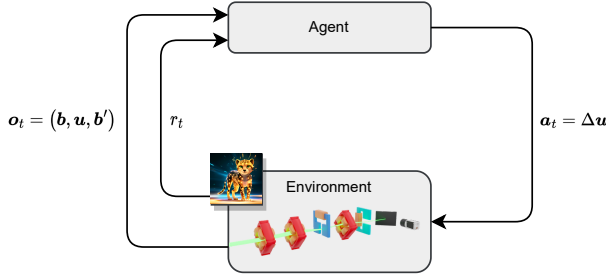
FIG. 3. Flowchart of the RL loop for the ARES EA transverse tuning task. The environment – during training defined in terms of Cheetah – outputs an observation $\boldsymbol{o}_t$ and a r_t based on the previous action $\boldsymbol{a}_{t-1}$. The agent then computes a new action $\boldsymbol{a}_t$ using the neural network policy. The new action is applied to the environment and results in the next observation $\boldsymbol{o}_{t+1}$ and reward r_{t+1}.

(TD3) [43] algorithm is used for its relative training sample efficiency among model-free RL algorithms. Specifically, we employ the implementation provided by the *Stable Baselines3* [44] Python package. As originally introduced to the field of RL for accelerators in [1], domain randomisation [32] is performed during training by sampling magnet and screen misalignments, as well as the incoming beam parameters and the target beam from a uniform distribution at the start of each rollout episode. We define the reward for transverse beam parameter tuning as

$$R\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) = \begin{cases} \hat{R}\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) & \text{if } \hat{R}\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) > 0 \\ 2 \cdot \hat{R}\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) & \text{otherwise} \end{cases} \quad (5)$$

with $\hat{R}\left(\boldsymbol{s}_t, \boldsymbol{a}_t\right) = O\left(\boldsymbol{u}_t\right) - O\left(\boldsymbol{u}_{t+1}\right)$, where the objective function $O\left(\boldsymbol{u}_t\right)$ is the logarithmic and weighted difference between the observed and target beams

$$O\left(\boldsymbol{u}_t\right) \coloneqq \ln \frac{1}{4} \sum_{i=1}^{4} \boldsymbol{w}^{(i)} \left| \boldsymbol{b}^{(i)} - \boldsymbol{b}'^{(i)} \right|. \quad (6)$$

A weight vector $\boldsymbol{w} = (1, 2, 1, 2)$ was chosen for the final training.

The policy is trained over a total of 6 000 000 interactions with the Cheetah environment. One interaction with the real environment involves sending new set points to the magnet power supplies, waiting for the power supplies to finish ramping to their set points, and taking multiple images of the diagnostic screen with the beam turned on and with the beam turned off. Altogether this process takes ca. 10 s to 20 s. Consequently, training on the real ARES accelerator would take about 3 years of continuous beam time. With beam time a scarce resource, such a long training is infeasible. Fast simulations like Cheetah can be computed faster than real time and enable us to collect the equivalent of many years of real time experience in much more feasible time frames. As was shown in Section II D, Cheetah is especially fast, allowing for the equivalent of 3 years of experience to be collected in just 27 min. Other RL algorithms such as Proximal Policy Optimisation (PPO) also allow for parallel rollouts on multiple environments. Using a simulation like Cheetah means that experience could be collected even faster in this case, while we usually do not have access to multiple of the same accelerator for training.

Despite having been trained using a comparatively simple simulation and deployed to the real world zero-shot without fine-tuning, our RL policy successfully tunes the transverse beam parameters on the real accelerator, finding magnet settings that achieve beam parameters closer to the target than those found by other state-of-the-art black-box optimisation algorithms. Moreover, the trained policy converges on these magnet settings in just a few samples, tuning the beam in less wall time than human operators while achieving comparable results. An
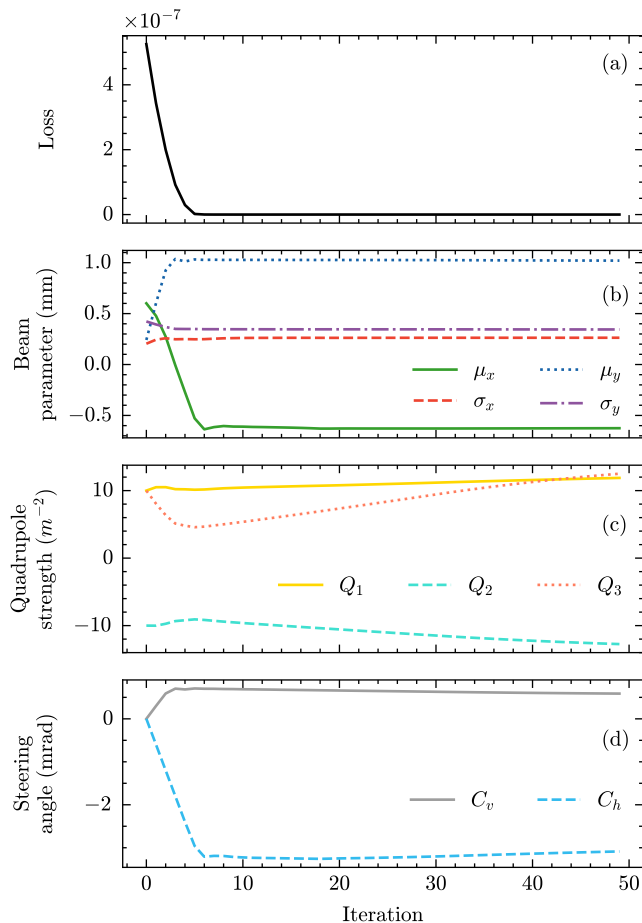
FIG. 4. An NN policy trained with RL tuning of the transverse beam parameters in the ARES EA. (a) The MSE loss development over parameter update iterations. (b) Beam parameters on the diagnostics screen over parameter update iterations. (c) and (d) quadrupole and dipole magnet settings over parameter update iterations.

example of a trained policy from [5] tuning the transverse beam parameters in the EA is shown in Fig. 4. Here it can be observed that the target transverse beam parameters of $\boldsymbol{b}' = (-0.61\,\text{mm}, 0.26\,\text{mm}, 1.03\,\text{mm}, 0.35\,\text{mm})$ are reached after about 6 steps. For more detailed results and discussions, please refer to [1, 5].

B. Gradient-based beam tuning and lattice optimisation

In the field of particle accelerators, there are various optimisation tasks, ranging from lattice optimisation in the design phase of a facility to tuning actuators at run time. In some cases, these tasks have too many degrees of freedom to be feasibly solvable by black-box optimisation algorithms, such as BO or RLO. However, their underlying function and its parameters may be known. In such cases, gradient-based optimisation may be used.

The latter has well-understood convergence properties and extensive tooling for it has been developed in the field of ML. Using gradient-based tuning on a model of an accelerator can help find good setups without the need for beam time. Even in cases where there exists a model mismatch, this offline optimisation approach can provide good starting points close to the optimum, which can then be further optimised online. Further, Cheetah can be used to reduce model mismatches through gradient-based system identification as is described in Section III C.

In this example, we consider the same transverse beam tuning task as in Section III A. In contrast to before, we assume that unobserved properties, such as the incoming beam and the beamline components' misalignments, are known. This may be the case in simulations during the design stage of an accelerator, if an accelerator is known to deviate very little from its design parameters, or if system identification like in Section III C was performed ahead of time.

The ARES lattice is loaded as a Cheetah `Segment`. Because Cheetah defines segments as PyTorch `Module`, all that is needed for PyTorch to automatically compute the gradient of the ARES EA with respect to the five magnet settings, is to define the latter as PyTorch `Parameter`. A fixed incoming beam is tracked through the EA Cheetah `Segment`. The resulting beam parameters can then be read from the diagnostic screen station at its end, and a MSE loss, defined as

$$\frac{1}{4} \sum_{i=1}^{4} \left(\boldsymbol{b}^{(i)} - \boldsymbol{b}'^{(i)} \right)^2, \tag{7}$$

with $\boldsymbol{b}'$ being the target beam and $\boldsymbol{b}$ the currently observed beam at the screen station, can be computed, where $\boldsymbol{b}^{(i)}$ denotes the i-th element of $\boldsymbol{b}$. PyTorch's automatic differentiation features can then be used to compute the gradient of the particle tracking and transverse beam parameter loss function with respect to the magnet settings

$$\boldsymbol{b} = f_{\text{EA}} \left(\boldsymbol{u} \mid \boldsymbol{m}, \boldsymbol{b}_{in} \right). \tag{8}$$

Adam [45], a variant of stochastic gradient descent (SGD) is then used to compute the updates to the magnet settings based on the computed gradient.

However, as is, this simple setup would result in unstable convergence. This is because the magnet settings are on very different scales, with the maximum quadrupole setting at $72\,\text{m}^{-2}$ and a maximum dipole magnet setting of $6.2\,\text{mrad}$. To address this, the magnet settings are normalised, i.e. scaled to normally fall into the range $[-1, 1]$. With Cheetah, this is easily achieved by wrapping the ARES EA `Segment` in an outer PyTorch `Module` with the normalised magnet settings as the PyTorch `Parameter`. On every call to the module's `forward` method, the segment's magnet settings $\boldsymbol{u}$ are set to
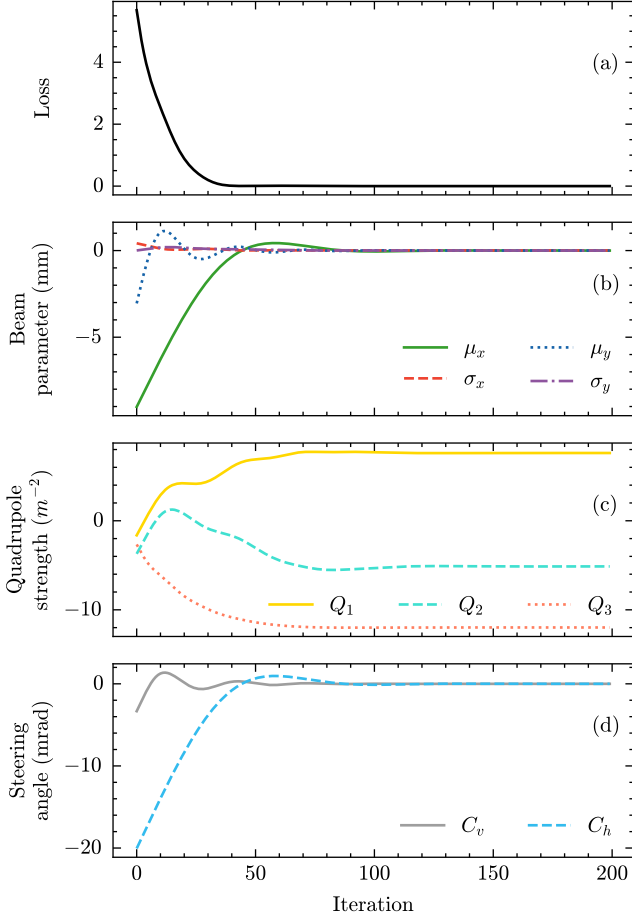
FIG. 5. Gradient-based tuning example of the transverse beam parameters in the ARES EA. (a) The loss development over parameter update iterations. (b) Beam parameters on the diagnostics screen over parameter update iterations. (c) and (d) quadrupole and dipole magnet settings over parameter update iterations.

$$\boldsymbol{u} = \boldsymbol{u}_{\text{normed}} \odot \boldsymbol{\lambda}, \qquad (9)$$

i.e. the element-wise product of the normalised actuator parameters $\boldsymbol{u}_{\text{normed}}$ and the scaling factors for each respective actuator component $\boldsymbol{\lambda}$. For the presented case study we use $\boldsymbol{\lambda} = \left(5\,\text{m}^{-2}, 5\,\text{m}^{-2}, 6.2\,\text{mrad}, 5\,\text{m}^{-2}, 6.2\,\text{mrad}\right)$. Note that for the quadrupole magnets, the scaling factors are chosen to be smaller than the physical limits of the real magnets so that they represent the commonly used operational ranges of these magnets at ARES more accurately.

The resulting convergence of the magnet settings can be seen in Fig. 5. In the shown example, the target beam parameters are $\boldsymbol{b}' = (0.0\,\mu\text{m}, 0.0\,\mu\text{m}, 0.0\,\mu\text{m}, 0.0\,\mu\text{m})$. We observe that the final magnet settings result in the desired centred and focused beam. The absolute deviation of the observed transverse beam parameters to the

target transverse beam parameters is $|\Delta\mu_x| = 0.33\,\mu\text{m}$, $|\Delta\sigma_x| = 6.66\,\mu\text{m}$, $|\Delta\mu_y| = 0.07\,\mu\text{m}$, and $|\Delta\sigma_y| = 0.85\,\mu\text{m}$, resulting in a mean absolute error (MAE) of $1.98\,\mu\text{m}$. Convergence is generally smooth, with all five magnets converging on their final settings after about 90 gradient steps. Note that the hyperparameters for this example were not tuned and better results may be possible.

C. Gradient-based system identification and virtual diagnostics

A common challenge with accelerators is that some properties of the beam or the accelerator hardware itself are not observable. Finding these properties usually requires multiple samples at different system states, ideally collected in a structured manner such as a grid scan for best results. Using these samples to reconstruct the hidden properties constitutes an inverse problem. Inverse problems are notoriously difficult to solve. Performing structured measurements to identify hidden properties of an accelerator necessitates an interruption of beam delivery, making it a costly measurement that is only performed if strictly needed.

Here we consider a system identification task in the ARES EA. There are two unknowns in the EA: The incoming beam and the misalignments of the quadrupoles. For this example, we aim to identify the misalignments of the quadrupoles. Knowing these can help tune the accelerator, for example by inserting the found misalignments into the model used in Section III B, or by using them to better align the quadrupoles and thereby reduce the dipole effect they have on the beam when used for focusing.

The gradients are computed with respect to the misalignments, i.e. the horizontal and vertical displacements of the magnets. Similar to the tuning example, the misalignments are normalised by wrapping the EA `Segment` in a PyTorch Module that holds normalised misalignment $\boldsymbol{m}_{\text{normed}}$ Parameters such that

$$\boldsymbol{b} = f_{\text{EA}}\left(\boldsymbol{m}_{\text{normed}}|\boldsymbol{b}_{in}, \boldsymbol{u}\right). \qquad (10)$$

The resulting optimisation problem is defined as

$$\min_{\boldsymbol{m}} O\left(\boldsymbol{m}\right) = \left(\mu_x - \mu_x'\right) + \left(\mu_y - \mu_y'\right) \qquad (11)$$

to find the misalignments such that, when these misalignments are assumed in the model, the beam positions predicted on the screen best match the experimental measurements. For this example, we use parasitically acquired measurements from other unrelated experiments. This effectively enables zero-shot system identification. The specific data used here was collected during evaluations of the RLO policies in [5], which was also referenced in the example in Section III A.

We first start with data collected in simulations, where the misalignments are known. This allows us to verify
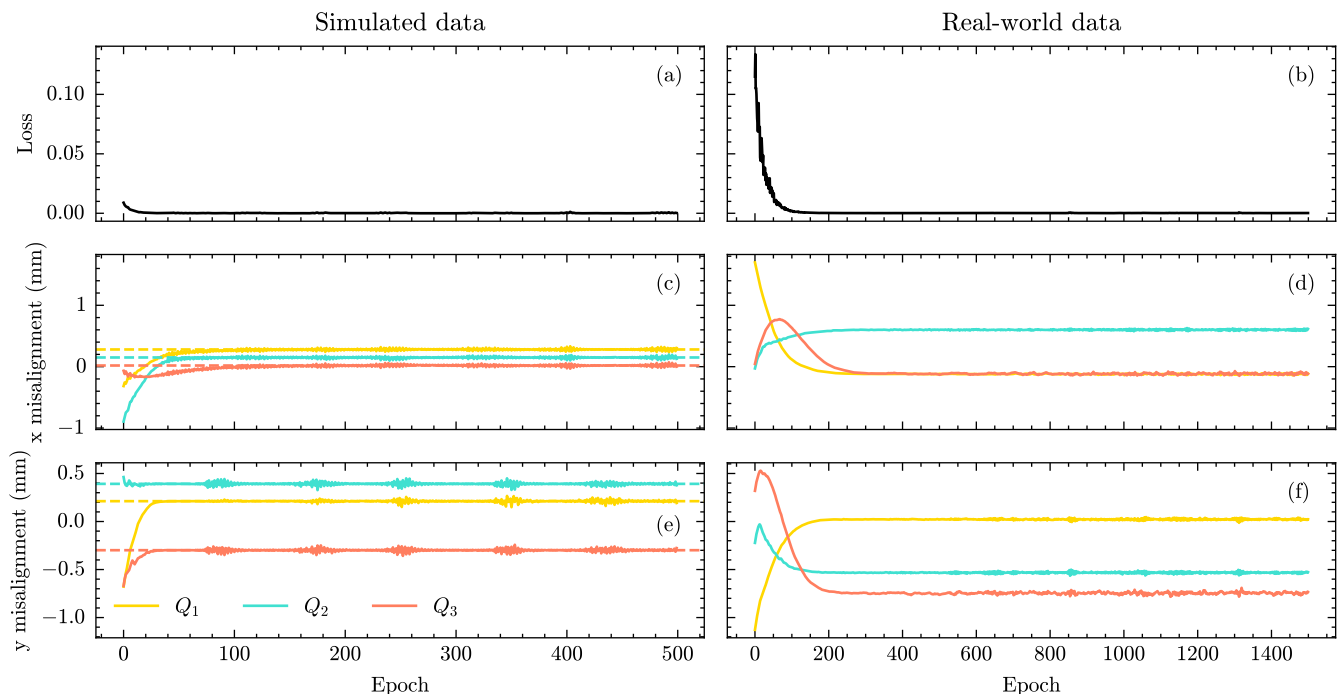
FIG. 6. Two examples of gradient-based quadrupole misalignment identification. For the first example, using parasitically collected simulated data, we show in (a) the loss, (c) the horizontal misalignments, and (e) the vertical misalignments over epochs. Dashed lines signify the ground truth misalignments, which are known in simulation. We show in (b) the loss, (d) the horizontal misalignments, and (f) the vertical misalignments for the second example using real-world parasitically collected data.

that the gradient-based optimisation arrives at correct misalignments. Because we are considering simulated data, the correct incoming beam is also known and may be assumed in the Cheetah model. As can be seen in Fig. 6 (a), (c), and (e), the reconstruction arrives at the correct misalignments.

Now trusting the misalignment reconstruction, the latter may be tried on data collected from a real-world experiment. With real-world data, however, the incoming beam is often unknown. Therefore, a sensible assumption on the incoming beam must be made. In the ARES EA, the goal is to reduce the misalignments of the quadrupole magnets with respect to the design orbit, which is the centre of the beam pipe in most cases. We therefore assume that the incoming beam position and momentum are at zero, i.e. we consider the observed orbit to be the design orbit. Doing so effectively sets the origin of the Cheetah model to the observed orbit in the data, resulting in misalignments being measured as deviations from that orbit. Other properties such as the beam size and energy only marginally affect this particular system identification setup and are therefore not considered. As can be seen in Fig. 6 (b), (d), and (f), the reconstruction appears to also perform well and smoothly arrive at sensible results under these conditions. However, the results cannot be checked against the ground truth this time, because the ground truth cannot be known.

D. Physics-based prior mean for Bayesian optimisation

Thanks to its speed, Cheetah can provide fast predictions of the beam parameters and guide optimisation algorithms during online tuning tasks, ultimately boosting their performance. One particular use case is BO, which utilises a Gaussian process (GP) model to build a surrogate model of the observed data and efficiently optimise the objective function. However, when dealing with high-dimensional tasks, BO tends to over-explore the parameter space to find the global optimum, inevitably increasing the required number of samples [46]. This limits the tasks which are solvable with classical BO algorithms to those that have less than a few dozen of input parameters [13]. Recent studies have shown that the convergence speed of BO can be significantly improved by incorporating prior knowledge of the accelerator system into the GP model, for example by including the correlation of quadrupole magnets into the GP covariance [47] or using an NN surrogate model as the prior mean function for the GP [48]. In the case of an NN surrogate, it should be accurate enough, as a wrong prior will only hamper the performance of BO. However, training such an NN model requires many samples either from simulation or real measurements, which are often not readily available. Cheetah becomes a promising alternative for the prior mean function for BO due to its fast inference
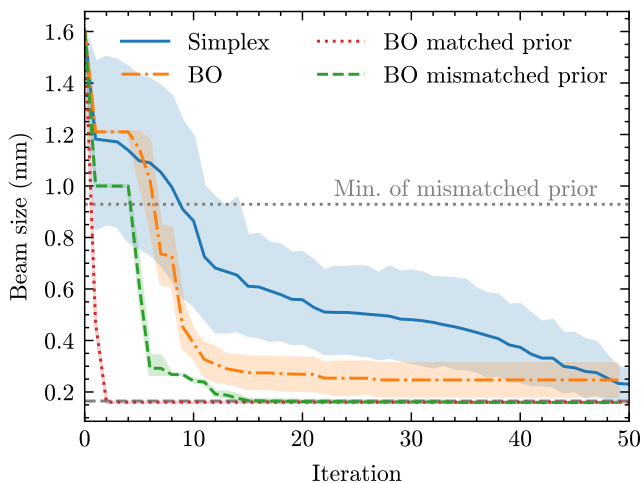
FIG. 7. Optimisation results of a beam focusing task using Nelder-Mead simplex (blue), Bayesian optimisation with a constant mean function (orange), and BO with a Cheetah `Segment` as the prior mean matched to the task (red) or mismatched (green). The results are averaged over 10 runs for each algorithm and shaded regions represent one standard deviation. The dotted line shows the minimum objective of the prior mean in the mismatched lattice and the dashed grey line denotes the true minimum for each task obtained from grid scans.

time. In addition, Cheetah's differentiability allows efficient acquisition function optimisation using gradient descent methods in modern BO packages like *BoTorch* [49].

In the following, we demonstrate the usage of Cheetah as a prior mean for BO in a beam-focusing task. The investigated lattice segment is a FODO cell consisting of two quadrupole magnets $\{Q_1, Q_2\}$ and a diagnostic screen at the end. The objective is to minimise the mean beam size measured at the screen

$$O := \frac{1}{2}(|\sigma_x| + |\sigma_y|) \tag{12}$$

by changing the quadrupole strengths $\{k_{Q_1}, k_{Q_2}\}$, where σ_x and σ_y denote the horizontal and vertical beam sizes respectively.

We evaluated the performance of BO with a Cheetah simulation model as a prior mean function and used BO with a constant prior mean and Nelder-Mead simplex as baselines. All algorithms are implemented based on the *Xopt* package [50]. Both BO variants use the Matérn-5/2 kernel and upper confidence bound (UCB) acquisition function with $\beta = 2.0$, which is a standard choice of hyperparameters for BO applications. Each algorithm was repeated 10 times starting from the same detuned setting and the averaged results with one standard deviation are shown in Fig. 7. In the case of the prior mean matched to the tuning task, BO with prior could immediately find the global minimum without exploring much of the parameter space. We then changed the lattice distances so that the BO prior mean is mismatched

to the ground truth of the tuning task. BO with prior first sampled around the minimum predicted by the prior mean, which is denoted by the dotted line in Fig. 7(b). Since there was a difference between the predicted and observed beam sizes, it continued exploring the parameter space and subsequently converged to the minimum.

In both cases, BO with prior converged to the true minimum within 15 steps and was more sample-efficient than BO with a constant mean prior and simplex algorithm even for the two-dimensional task. This is expected to have a larger impact on higher-dimensional tasks as the parameter space grows exponentially. Furthermore, when BO with a Cheetah prior mean is applied to real-world tasks, one can use system identification to determine the mismatch between the simulated and the real accelerator using the obtained data parasitically, as shown in Section III C. This allows further reduction of the discrepancy between the physics-based prior mean and the real-world systems.

E. Integrating modular neural network surrogates with beam dynamics simulations

Some beam dynamics such as collective effects require expensive computations to simulate. This problem has previously been solved using NN surrogate models. Data from experiments or high-fidelity simulations can be used to train an NN to approximate the real world or a high-fidelity simulation with a high degree of accuracy, while forward passes of NNs are cheap to compute. To date, single NN surrogates are usually used to model, for example, specific instruments or lattice setups [2, 4, 25]. As a result, these models have limited versatility and reusability, and novel applications require the design and training of new models, which necessitates further beam time or computation to acquire new training data sets.

Modular surrogate models over all parameters of generic accelerator elements can be used as a versatile, reusable, and reconfigurable approach to modelling larger parts of accelerators using NNs. This modular approach integrates well with Cheetah, as is shown in Fig. 1 (d). Modular NN surrogates computing expensive physical effects can seamlessly be wrapped as Cheetah elements and combined with other elements using beam dynamics models already implemented in Cheetah or other NN surrogates. Crucially, NNs are differentiable and commonly implemented in PyTorch. Hence, they integrate well with Cheetah's PyTorch backend and preserve PyTorch's automatic differentiation functionality.

In this use case example, we demonstrate the implementation of a quadrupole magnet augmented with space charge effects modelled using a NN and integrate it as an element in Cheetah. The straightforward implementation of an NN-based modular surrogate model would see the `track` method in Cheetah implemented as a forward pass of an NN
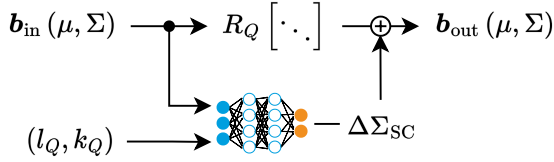
FIG. 8. Scheme of the NN-enhanced quadrupole module. The incoming `ParameterBeam` P is multiplied with the magnet's transfer matrix R_Q as in classical linear beam dynamics. The NN model predicts changes $\Delta\Sigma_{\mathrm{SC}}$ in beam parameters due to the space charge effects, resulting in the outgoing beam $\boldsymbol{b}_{\mathrm{out}}$.

$$\boldsymbol{b}_{\mathrm{out}} = f_{\mathrm{SC}}\left(\boldsymbol{b}_{in}|l_Q, k_Q\right), \tag{13}$$

mapping the incoming beam $\boldsymbol{b}_{in}$ and quadrupole parameters (l_Q, k_Q) to an outgoing beam $\boldsymbol{b}_{\mathrm{out}}$. Fortunately, the effects of space charge in a quadrupole are secondary to the linear beam dynamics, and linear beam dynamics can be modelled easily. To reduce the complexity of the function modelled by the NN and reduce the time and data required to train it, the tracking function through the quadrupole element is instead reformulated as

$$\boldsymbol{b}_{\mathrm{out}} = f_{\mathrm{linear}}\left(\boldsymbol{b}_{in}|l_Q, k_Q\right) + \Delta\Sigma_{\mathrm{SC}}\left(\boldsymbol{b}_{in}|l_Q, k_Q\right), \tag{14}$$

where $f_{\mathrm{linear}}\left(\boldsymbol{b}_{in}|l_Q, k_Q\right)$ is the handcrafted computation of the linear beam dynamics through a quadrupole magnet already implemented by Cheetah and $\Delta\Sigma_{\mathrm{SC}}\left(\boldsymbol{b}_{in}|l_Q, k_Q\right)$ is the change induced to the outgoing beam by space charge effects. The NN model is used to approximate $\Delta\Sigma_{\mathrm{SC}}\left(\boldsymbol{b}_{in}|l_Q, k_Q\right)$. An illustration of this process is provided in Fig. 8.

Data for training the modular NN surrogate model is generated using Ocelot [10] with space charge effects and second-order tracking through a single quadrupole element. Space charge effects are calculated with a mesh size of 63^3 and applied at a unit step size of $2\,\mathrm{cm}$. A total of $100\,000$ samples are collected from uniform distributions over length l_Q and strength k_Q of the quadrupole, as well as log-uniform distributions over a subset of the incoming beam parameters

$$\boldsymbol{x} \coloneqq (\sigma_x, \sigma_{x'}, \sigma_y, \sigma_{y'}, \sigma_\tau, \sigma_\delta, E, q) \subset \boldsymbol{b}_{in}, \tag{15}$$

where E is the beam energy and q is the total charge of the beam. The ranges over which these are sampled for data generation are shown in Table II. A log-uniform distribution was chosen for the beam input parameters because their order of magnitude is more relevant in space charge computations.

The neural network model takes $\boldsymbol{x}$ as input and outputs

$$\boldsymbol{y} \coloneqq (\Delta\sigma_x, \Delta\sigma_{x'}, \Delta\sigma_y, \Delta\sigma_{y'}, \Delta\sigma_\tau, \Delta\sigma_\delta) \tag{16}$$

TABLE II. Input parameter ranges for data set generation

Input parameter	Range
σ_x	$[10\,\mu\mathrm{m}, 1\,\mathrm{mm}]$
$\sigma_{x'}$	$[10\,\mu\mathrm{rad}, 1\,\mathrm{mrad}]$
σ_y	$[10\,\mu\mathrm{m}, 1\,\mathrm{mm}]$
$\sigma_{y'}$	$[10\,\mu\mathrm{rad}, 1\,\mathrm{mrad}]$
σ_τ	$[300\,\mathrm{nm}, 300\,\mu\mathrm{m}]$
σ_δ	$[10^{-5}, 10^{-3}]$
q	$[1\,\mathrm{pC}, 5\,\mathrm{nC}]$
E	$[1\,\mathrm{MeV}, 1\,\mathrm{GeV}]$

TABLE III. NN training hyperparameters

Hyperparameter	Value
Batch size	32
Hidden activation	Sigmoid
Hidden layer width	256
Learning rate	2×10^{-5}
Number of epochs	959 (max. $10\,000$)
Number of hidden layers	4
Gradient descent algorithm	Adam

the changes to the beam parameters resulting from space charge effects when compared to linear beam dynamics, such that

$$\Delta\Sigma_{\mathrm{SC}}\left(\boldsymbol{b}_{in}|l_Q, k_Q\right) \approx \boldsymbol{y} = f_{\mathrm{NN}}\left(\boldsymbol{x}\right). \tag{17}$$

We choose an MLP architecture for the NN and the Adam [45] gradient descent algorithm for adjusting the parameters of the model. Early stopping with a *patience* (number of steps with no improvement before the training is terminated) of 10 was used. The data set is split $60/20/20$ into training, validation, and test sets. The logarithm is taken of all beam parameter inputs before they are input into the NN model. All inputs and outputs are scaled to fit a unit-normal distribution with scaling on the beam parameter inputs performed after taking the logarithm. Hyperparameters were tuned over a total of 303 trainings using Bayesian optimisation, with *PyTorch Lightning* [51] used to implement the training and *Weights & Biases* [52] for experiment tracking. The best-observed hyperparameters used for the final model are listed in Table III.

Figure 9 shows the beam sizes, beam divergence, bunch length, and energy spread over different beam energies as computed by the NN-augmented quadrupole implemented in Cheetah, and compares them to Cheetah's default linear beam dynamics tracking and Ocelot's space charge simulation. All other parameters of the incoming beam and the quadrupole parameters are fixed. We observe that the NN-augmented quadrupole implementation correctly infers larger effects of space charge at low energies when compared to linear beam dynamics simulations without space charge effects. Congruently, the beam parameters computed using the NN-augmented quadrupole in Cheetah closely match those computed us-
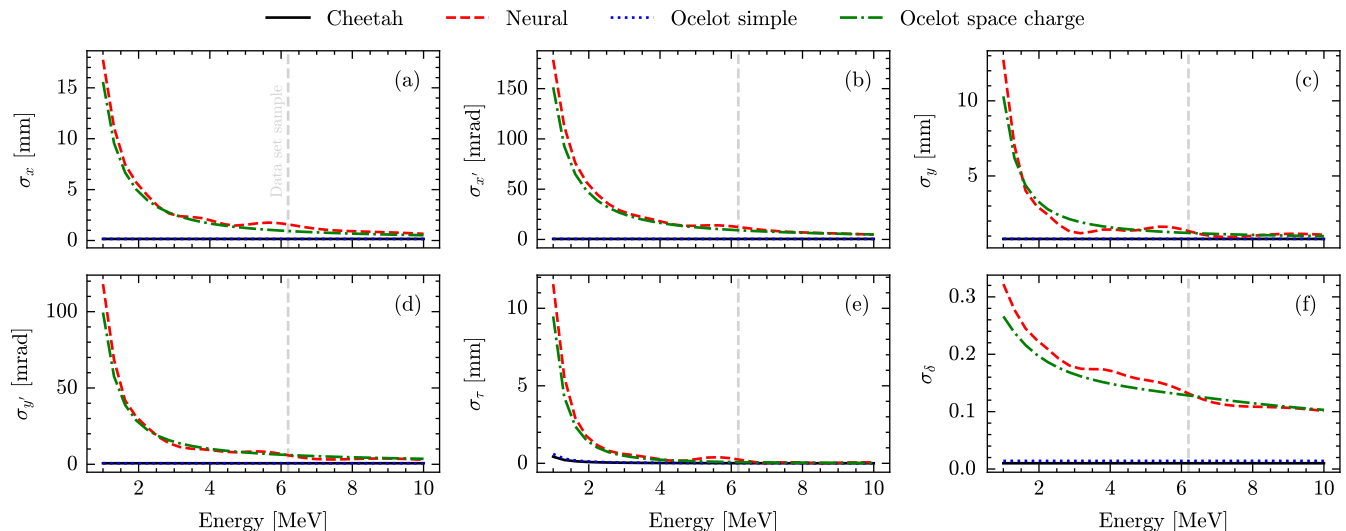
FIG. 9. Outgoing beam parameters for the same incoming beam over different energies tracked with the default linear quadrupole in Cheetah (black), the NN-augment space charge quadrupole (red), Ocelot without space charge calculations (blue), and Ocelot with space charge calculations (green).

ing Ocelot with space charge effects, indicating that our simulations accurately capture space charge effects.

We benchmark the computational speed of the NN-augmented space charge quadrupole element against Ocelot's space charge simulation. For this evaluation, the incoming beam and quadrupole settings considered for the experiments in Fig. 9 are used with an energy of 6.2 MeV. Ocelot is configured with the same space charge simulation setup as was used to generate the training data. We find that Ocelot takes an average of 1.36 s to perform space charge tracking through a single quadrupole, while the same simulation is performed by the NN-augmented Cheetah element in an average of 370 μs – a reduction in compute time of more than 3 orders of magnitude. These benchmarks were run on the same Apple M1 Pro SOC's CPU considered in Section II D. Unlike Ocelot's space charge implementation, Cheetah can take advantage of hardware acceleration on GPU, which is expected to result in a further reduction of compute times.

Moreover, because an NN is used for the modular surrogate modelling, the computation remains fully differentiable, in effect providing differentiable space charge simulations that can seamlessly be integrated with other beam dynamics simulations.

This example serves as a proof of concept for Cheetah's ability to provide a platform for modular surrogate modelling. In fact, to the best of our knowledge, this is the first instance of modular NN-based surrogate modelling in particle accelerator simulations. As such, the integration of modular NN surrogate models in Cheetah enables us to build data-driven, high-speed, high-fidelity models of beam dynamics that would otherwise require computationally expensive models. More-

over, Cheetah can also be integrated with modular NN surrogates trained on real-world data, allowing for mitigation of model mismatches. In future work, we hope to add more NN-augmented and fully NN-based elements to Cheetah. While the presented example applies to parametrically defined beams, it can easily be extended to beams defined as particle clouds by employing NN architectures such as *PointNet* [53], which is intended as a future extension of Cheetah.

IV. CONCLUSIONS

In this work, we introduced Cheetah, a Python package providing high-speed differentiable beam dynamics simulations for machine learning applications. Cheetah is easy to use, provides an extensible platform for future differentiable models and integrates well with the ML ecosystem in Python. Moreover, we demonstrated Cheetah's capabilities using five example applications. We illustrated its speed training a NN policy to perform transverse beam tuning while achieving zero-shot transfer to the real world. Further, we showed how automatic differentiation in Cheetah can be used for gradient-based beamline tuning as well as gradient-based system identification. Cheetah's usefulness as a differentiable prior for Bayesian optimisation was also shown while optimising beam focusing through a FODO cell. Lastly, we presented an example of how Cheetah can easily be extended by training a modular NN space charge model to predict how space charge affects the beam when tracked through a single quadrupole magnet.

Cheetah will see continued extension as a tool for our work. We further hope that in the future, members of

the community will collaborate in extending Cheetah, for example with already developed differentiable models of processes in particle accelerators. Such collaboration and integration will help make these tools more accessible to the community. Further extensions of Cheetah planned from our side include the integration of additional modular surrogate models, in particular for single particle tracking based on PointNet [53], and batched parallel execution of simulations to increase speeds further. In addition, experiments using JAX [17] as a backend for Cheetah might aid in attaining even faster simulation speeds as was seen by switching the backend of Stable Baselines3 to JAX in SBX [44].

CODE AVAILABILITY

The source code of Cheetah is hosted at `https://github.com/desy-ml/cheetah`. The code for the presented use case examples is available from `https://github.com/desy-ml/cheetah-demos`. More extensive code regarding the RL example may be found at `https://github.com/desy-ml/rl-vs-bo`.

ACKNOWLEDGMENTS

AUTHOR CONTRIBUTIONS

J.K. originally conceived and developed Cheetah as a fast simulation code. Further development of Cheetah as a differentiable simulation code was done by J.K. and C.X. J.K. undertook example studies of Cheetah on reinforcement learning, gradient-based tuning, gradient-based system identification, and modular neural network surrogate modelling. C.X. performed example studies on using Cheetah as a prior for Bayesian optimisation and contributed to the other examples studies. J.K. wrote the manuscript. C.X. contributed sections to the manuscript and provided substantial edits. A.E. and A.S.G. secured funding. All authors read and edited the manuscript.

[1] J. Kaiser, O. Stein, and A. Eichler, in *Proceedings of the 39th International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 162, edited by K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato (PMLR, 2022) pp. 10575–10585.

[2] A. L. Edelen, S. G. Biedron, J. P. Edelen, S. V. Milton, and P. J. V. D. Slot (JACoW Publishing, 2017) pp. 488–491.

[3] K. Fujita, IEEE Access **9**, 164017 (2021).

[4] J. Kaiser, A. Eichler, S. Tomin, and Z. Zhu (2023).

[5] J. Kaiser, C. Xu, A. Eichler, A. S. Garcia, O. Stein, E. Bründermann, W. Kuropka, H. Dinter, F. Mayet, T. Vinatier, F. Burkart, and H. Schlarb, Learning to do or learning while doing: Reinforcement learning and bayesian optimisation for online continuous tuning (2023), arXiv:2306.03739 [cs.LG].

[6] C. Xu, J. Kaiser, E. Bründermann, A. Eichler, A.-S. Müller, and A. Santamaria Garcia, in *Proceedings of the 14th International Particle Accelerator Conference*, 14 (JACoW Publishing, Geneva, Switzerland, 2023) pp. 4487–4490.

[7] V. Kain, S. Hirlander, B. Goddard, F. M. Velotti, G. Z. Della Porta, N. Bruchon, and G. Valentino, Physical Review Accelerators and Beams **23**, 124801 (2020).

[8] J. Degrave, F. Felici, J. Buchli, M. Neunert, B. Tracey, F. Carpanese, T. Ewalds, R. Hafner, A. Abdolmaleki, D. Casas, C. Donner, L. Fritz, C. Galperti, A. Huber, J. Keeling, M. Tsimpoukelli, J. Kay, A. Merle, J.-M. Moret, and M. Riedmiller, Nature **602**, 414 (2022).

[9] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, Nature **575** (2019).

[10] I. Agapov, G. Geloni, S. Tomin, and I. Zagorodnov, Nuclear Instruments and Methods in Physics Research **768** (2014).

[11] S. Tomin, I. Agapov, M. Dohlus, and I. Zagorodnov, in *Proc. of International Particle Accelerator Conference (IPAC'17), Copenhagen, Denmark, 14-19 May, 2017*, International Particle Accelerator Conference No. 8 (JACoW, Geneva, Switzerland, 2017) pp. 2642–2645.

[12] Z. Zhang, A. Edelen, C. Mayes, J. Garrahan, J. Shtalenkova, R. Roussel, S. Miskovich, D. Ratner, M. Boese, S. Tomin, G. Wang, and Y. Hidaka, in *Proceedings of the 13th International Particle Accelerator Conference (IPAC 2022)* (2022).

[13] R. Roussel, A. L. Edelen, T. Boltz, D. Kennedy, Z. Zhang, X. Huang, D. Ratner, A. S. Garcia, C. Xu, J. Kaiser, A. Eichler, J. O. Lubsen, N. M. Isenberg, Y. Gao, N. Kuklev, J. Martinez, B. Mustapha, V. Kain, W. Lin, S. M. Liuzzo, J. S. John, M. J. V. Streeter, R. Lehe, and W. Neiswanger, Bayesian optimization algorithms for accelerator physics (2023), arXiv:2312.05667 [physics.acc-ph].

[14] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. Michael, S. Ranjan, S. Xiaoqing, E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, Llama 2: Open foundation and fine-tuned chat models (2023).

[15] G. Team, Gemini: A family of highly capable multimodal models (2023).

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, in *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) pp. 8024–8035.

[17] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, JAX: composable transformations of Python+NumPy programs (2018).

[18] K. Flöttmann, ASTRA – A space charge tracking algorithm (1997).

[19] D. Sagan, *Computational accelerator physics. Proceedings, 8th International Conference, ICAP 2004, St. Petersburg, Russia, June 29-July 2, 2004*, Nucl. Instrum. Meth. **A558**, 356 (2006), proceedings of the 8th International Computational Accelerator Physics Conference.

[20] M. Borland, in *Proceedings of the 6th International Computational Accelerator Physics Conference* (2000).

[21] CERN, MAD-X – Methodical accelerator design (1990).

[22] G. Iadarola, R. D. Maria, S. Lopaciuk, A. Abramov, X. Buffat, D. Demetriadou, L. Deniau, P. Hermes, P. Kicsiny, P. Kruyt, A. Latina, L. Mether, K. Paraschou, Sterbini, F. V. D. Veken, P. Belanger, P. Niedermayer, D. D. Croce, T. Pieloni, and L. V. Riesen-Haupt, Xsuite: An integrated beam physics simulation framework (2023).

[23] J. G.-A. et al., in *Proc. IPAC'23*, IPAC'23 - 14th International Particle Accelerator Conference No. 14 (JACoW Publishing, Geneva, Switzerland, 2023) pp. 2797–2800.

[24] T. Dorigo, A. Giammanco, P. Vischia, M. Aehle, M. Bawaj, A. Boldyrev, P. de Castro Manzano, D. Derkach, J. Donini, A. Edelen, *et al.*, Reviews in Physics , 100085 (2023).

[25] D. Ratner, F. Christie, J. Cryan, A. Edelen, A. Lutman, and X. Zhang, Optics express **29**, 20336 (2021).

[26] R. Roussel, A. Edelen, D. Ratner, K. Dubey, J. P. Gonzalez-Aguilera, Y. K. Kim, and N. Kuklev, Phys. Rev. Lett. **128**, 204801 (2022).

[27] R. Roussel and A. L. Edelen, Applications of differentiable physics simulations in particle accelerator modeling (2022).

[28] R. Roussel, A. Edelen, C. Mayes, D. Ratner, J. P. Gonzalez-Aguilera, S. Kim, E. Wisniewski, and J. Power, Phys. Rev. Lett. **130**, 145001 (2023).

[29] S. Gasiorowski, Y. Chen, Y. Nashed, P. Granger, C. Mironov, D. Ratner, K. Terao, and K. V. Tsang, Differentiable simulation of a liquid argon time projection chamber (2023).

[30] J. Qiang, Phys. Rev. Accel. Beams **26**, 024601 (2023).

[31] O. Stein, J. Kaiser, and A. Eichler, in *Proceedings of the 13th International Particle Accelerator Conference* (2022).

[32] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017) pp. 23–30.

[33] J. Kaiser, C. Xu, O. Stein, and F. Theilen, Cheetah, `https://github.com/desy-ml/cheetah` (2021).

[34] `https://cheetah-accelerator.readthedocs.io`.

[35] K. L. Brown, Adv. Part. Phys. **1**, 71 (1968).

[36] J. Rosenzweig and L. Serafini, Phys. Rev. E **49**, 1599 (1994).

[37] F. Andreas, LatticeJSON, `https://github.com/nobeam/latticejson` (2019).

[38] E. Panofski *et al.*, Instruments **5** (2021).

[39] F. Burkart, R. Aßmann, H. Dinter, S. Jaster-Merz, W. Kuropka, F. Mayet, and T. Vinatier, in *Proceedings of the 31st International Linear Accelerator Conference (LINAC'22)*, International Linear Accelerator Conference No. 31 (JACoW Publishing, Geneva, Switzerland, 2022) pp. 691–694.

[40] A. Eichler, F. Burkart, J. Kaiser, W. Kuropka, O. Stein, C. Xu, E. Bründermann, and A. Santamaria Garcia, in *12th International Particle Accelerator Conference* (JACoW Publishing, 2021) pp. 2182–2185.

[41] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, Gymnasium (2023).

[42] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, OpenAI Gym (2016).

[43] S. Fujimoto, H. van Hoof, and D. Meger, Addressing function approximation error in actor-critic methods (2018), preprint available at `https://arxiv.org/abs/1802.09477v3`.

[44] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, Journal of Machine Learning Research **22**, 1 (2021).

[45] D. P. Kingma and J. Ba, CoRR **abs/1412.6980** (2014).

[46] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, Proceedings of the IEEE **104**, 148 (2016).

[47] J. Duris, D. Kennedy, A. Hanuka, J. Shtalenkova, A. Edelen, P. Baxevanis, A. Egger, T. Cope, M. McIntire, S. Ermon, and D. Ratner, Physical Review Letters **124** (2020).

[48] C. Xu, R. Roussel, and A. Edelen, arXiv preprint arXiv:2211.09028 (2022).

[49] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, in *Advances in Neural Information Processing Systems 33* (2020).

[50] R. Roussel, A. Edelen, A. Bartnik, and C. Mayes, in *Proc. IPAC'23*, IPAC'23 - 14th International Particle Accelerator Conference No. 14 (JACoW Publishing, Geneva, Switzerland, 2023) pp. 4796–4799.

[51] W. Falcon and The PyTorch Lightning team, PyTorch Lightning (2019).

[52] L. Biewald, Experiment tracking with Weights and Biases (2020), software available from wandb.com.

[53] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, arXiv preprint arXiv:1612.00593 (2016).