Optimizing a superconducting radio-frequency gun using deep reinforcement learning

David Meier,[1,3] Luis Vera Ramirez,[1] Jens Völker,[1] Jens Viefhaus,[1,3] Bernhard Sick,[2,3] and Gregor Hartmann[1,3]

[1] *Helmholtz-Zentrum für Materialien und Energie, Hahn-Meitner-Platz 1, 14109 Berlin, Germany*
[2] *Intelligent Embedded Systems, University of Kassel, Wilhelmshöher Allee 73, 34121 Kassel, Germany*
[3] *Artificial Intelligence Methods for Experiment Design (AIM-ED), Joint Lab Helmholtz-Zentrum für Materialien und Energie, Berlin (HZB) and University of Kassel, Hahn-Meitner Platz 1, 14109 Berlin, Germany*

(Received 21 March 2022; accepted 30 September 2022; published 28 October 2022)

Superconducting photoelectron injectors are promising for generating highly brilliant pulsed electron beams with high repetition rates and low emittances. Experiments such as ultrafast electron diffraction, experiments at the Terahertz scale, and energy recovery linac applications require such properties. However, optimizing the beam properties is challenging due to the high number of possible machine parameter combinations. This article shows the successful automated optimization of beam properties utilizing an already existing simulation model. To reduce the required computation time, we replace the costly simulation with a faster approximation with a neural network. For optimization, we propose a reinforcement learning approach leveraging the simple computation of the derivative of the approximation. We prove that our approach outperforms standard optimization methods for the required function evaluations given a defined minimum accuracy.

DOI: 10.1103/PhysRevAccelBeams.25.104604

I. INTRODUCTION

A superconducting radiofrequency (SRF) photoelectron injector is currently under construction as an electron source for the upcoming SRF accelerator SeaLab (formerly known as bERLinPro).

The SRF gun is a one-and-a-half cell SRF cavity with a photocathode at the back wall of the cavity. A pulsed extinction laser illuminates the photocathode and extracts electrons from the material. The standing wave radiofrequency field in the cavity accelerates the electrons to relativistic energies that travel toward the subsequent accelerator components, i.e., the focusing solenoid and the further SRF cavities. Due to the continuous and high repetition rate of the high brilliant electron beam, several experiments are planned, e.g., ultrafast electron diffraction, experiments employing Terahertz radiation, and energy recovery linac (linear accelerator) applications. More details on the SRF gun can be found in [1].

An accurate alignment of all components and optimal SRF gun configuration parameter settings are essential to achieve the necessary beam properties for the subsequent experiments. Fourteen parameters describe the geometry and radiofrequency parameters of the gun cavity, the position of the cathode, the position of the drive laser spot inside the cavity, and the alignment and magnetic parameters of the focusing solenoid. Unfortunately, only a few of them are measurable, and not all of them are adjustable. The correct alignment of these parameters significantly influences the performance of the complete accelerator and the operability of the downstream experiments.

In Fig. 1, we provide a schematic view of the components of the electron gun and the locations of the input and output parameters. With the first viewscreen approximately 1 m downstream of the SRF gun module, we can measure four parameters that describe the behavior of the extracted electron beam, such as the transverse position and the transverse beam size. Because these properties determine the quality of the resulting beam, we will minimize the horizontal and vertical beam size as well as center the horizontal and vertical beam position. We define this as our optimization task, which is addressed in this article.

To tackle this optimization task, we will use a technique from the area of machine learning called *reinforcement learning* (RL) [2,3]. The basic idea is that we have an algorithm, which we will call an *agent*, that influences an environment by performing actions on it. The agent decides to make an action a. We call the rule base used to make this

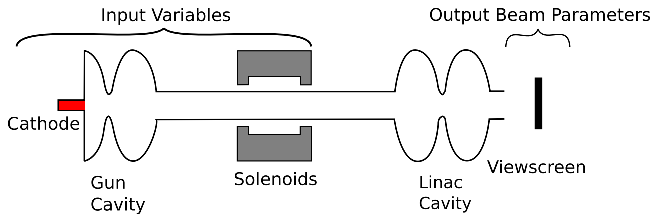

FIG. 1. Schematic view of the electron gun; the input variables are the parameters of the cathode, solenoids, and gun cavity. The output beam parameters can also be measured in the real device.

decision the *policy*. In our case, an action changes solenoid angles and positions, which impacts the electron beam. This action leads to changes in the environment. The environment is, in our case, a simulation approximation of the electron gun because both measurements in the real machine and the original simulation are too slow for training our agent. We define a reward function R_l that indicates how well action a is suited to achieve our optimization goal, i.e., to optimize the quality of the beam. To choose the following action, the agent gets a state variable s which provides information about the state of the environment. Based on the state and reward, the agent decides which, action to perform next [2]. We give a schematic view of the RL cycle in Fig. 2.

The main advances provided by the method proposed in this article are:

Fast inference requiring fewer reward evaluations. We compare different strategies for optimizing the beam properties to a particular level of accuracy. We expect our approach with RL to outperform other local optimization algorithms regarding the required reward evaluations, which we can equalize with computational time. Once we have trained our RL agent, we can change the parameters and quickly execute the optimization. This procedure differs from the previously used local optimization algorithms, which require several hundred thousand simulation evaluations.

Compound solution for the optimization task. In this article, we propose a compound pipeline for solving the optimization task of an electron gun. This method includes the solution of the offset finding task, which means finding the difference between the simulation and the real world of the input parameters and output screen variables. Our proposed pipeline is a considerable step toward automated self-optimizing the radiofrequency photoinjector.

Explainability of decisions. Furthermore, we will analyze the learned policy, which explains how the RL agent makes choices. This explainability makes the decisions more trustworthy. Moreover, the agent chooses actions from a limited interval of parameter values, which ensures the safe execution of these actions at all times.

In the next section, we will give an overview of the state-of-art of RL. Furthermore, we will present already existing approaches in applications of optimization tasks in electron gun and accelerator settings. In Sec. III, we will describe our proposed RL agent, which means we will define the optimized reward function and how our policy is updated. We will compare our approach with several local optimization algorithms in Sec. IV and give a conclusion and outlook in Sec. V.

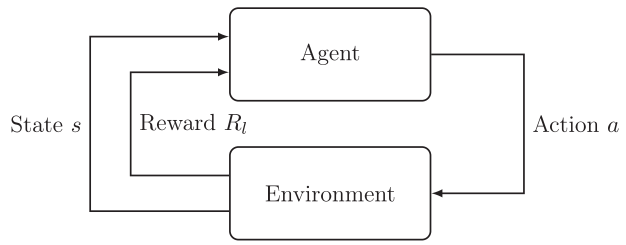

FIG. 2. Schematic representation of the RL cycle; The agent illustrates our RL algorithm. It can apply an action a to the environment. The environment depicts in our case, the simulation approximation which can calculate its new state s and the reward R_l. It gives this information to the RL agent to decide the next action.

II. RELATED WORK

We first briefly introduce RL based on [3] and then show several applications of machine learning and RL in a synchrotron context. The interested reader can find further details on RL in [2].

RL problems are modeled as *Markov decision processes*. That means we assume that the probability of a transition from one state s to another state s' depends only on s and not the predecessors of s. One of the most basic variants of RL is Q-Learning: It uses a table as a policy that contains the states and their Q values, which are the expected rewards for an action taken in a given state [2].

We calculate the *expected discounted return* to measure the performance of a policy: $R_t = \sum_{k=0} \gamma r_{t+k+1}$. The value $\gamma \in [0, 1)$ is a discount rate that weighs future rewards less strongly due to higher uncertainty. The reward obtained during the transition from s_t to s_{t+1} is denoted with r_{t+1}.

The policy table is updated according to the Bellman equation:

$$q_\pi(s, a) = \mathbb{E}_\pi(R_t | s_t = s, a_t = a). \tag{1}$$

It uses a stochastic policy function $\pi : S \times A \to [0, 1]$ with $\pi(s, a) = P(a|s)$. *Stochastic* means here that it is represented as a distribution of actions. In settings with a discrete action space, we can estimate the Q-value for each state-action pair. However, in continuous action space settings, this is not possible. *Policy gradient* methods learn the policy directly and thus can also map an input to continuous action spaces. The target is to maximize the objective function $J(\theta) = \mathbb{E}_{\pi_\theta}(R_t)$. That means we search for the parameters θ that maximize the expected discounted reward. A neural network can represent J, and the parameters θ are the weights of this neural network. Typically, we initialize the weights θ of neural networks randomly. The *stochastic policy gradient theorem* provides an estimate of the gradients the weights of the neural networks need to get updated to improve the occurring reward with the chosen actions. However, the stochastic policy gradient theorem depends on the unknown Q-value $q_\pi(s,a)$. We can approximate it by using the actual reward r_t after that action. This approach is called the REINFORCE learning rule [4]. Another way to solve this problem is to train a second neural network to approximate the Q value directly. This approach is called *actor-critic*, that means that the actor learns a policy π_θ only based on the state, whereas the critic learns to evaluate the Q value and gives this information to the actor again.

The deterministic policy gradient (DPG) method is actor-critic and learns a deterministic policy $\mu(s_t)$ [5]. Using a deterministic policy is advantageous because it does not have variability, and thus less training time is required. To still allow exploration, DPG uses an *off-policy* strategy, which means that a stochastic policy that differs from the learned policy chooses the taken actions. In [6], the authors developed an extension with deep neural networks and called it deep deterministic policy gradient (DDPG). It is model-free since it only depends on the gradient of the Q values. *Model-free* means that the algorithm does not depend on a function that predicts state transitions and rewards of the environment. DDPG has been successfully applied to many continuous control problems.

When an RL agent incorporates a neural network, the method is called *Deep RL* [7]. Similar applications of Deep RL approaches that we used in this study have already been successfully applied to different application scenarios in BESSY II [8], e.g., booster current, injection efficiency, and orbit correction. The method used in these scenarios uses DDPG.

Various methods have been proposed for local optimization and will serve as a reference in this study. We will compare the Nelder-Mead simplex algorithm [9], Powell's [10], and gradient descent [11] with our proposed RL approach.

Other approaches exist for optimizing an rf photoinjector with similar objective functions to the one used in this article. As a first step, [12] shows the use of a convolutional autoencoder that can compress the data in images. These images show the longitudinal phase space, which is the first step toward using images in a succeeding optimization algorithm. Another analysis of optimizing an rf photoinjector is using multiobjective Bayesian optimization [13]. The authors can tune an electron gun's parameters efficiently and show that they can find solutions sufficiently near the Pareto front of the beam optimization problem. In [14], the authors use also a surrogate model, i.e., a fast replacement for the simulation, comparable to the approach presented in this article. However, they use a genetic algorithm for optimization, which thus requires multiple hundreds of surrogate model evaluations. In [15], the authors propose to initialize the genetic algorithm with an invertible neural network that reduces the required amount to about ten surrogate model evaluations.

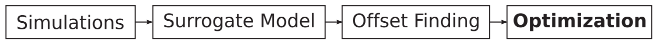

FIG. 3. Basic processing pipeline; this article focuses on the optimization part.

III. METHOD

We depict the general approach in Fig. 3. We first simulate the electron gun with randomly chosen parameters and create a data set of the outcomes. The next step is to learn a surrogate model as a faster replacement for the simulation. We use this model for the following offset finding and optimization steps. The focus of this study is the optimization part.

Surrogate model. We used the ASTRA (a space charge tracking algorithm) simulation for physical modeling [16]. It is physically precise but computationally intense. The assessment of one combination of parameters requires, on average, about 5 min of calculation time per core on a current CPU.

To overcome this issue, we trained a surrogate model, a neural network that replaces the simulation. The evaluation time of this surrogate model is on a scale of several hundred milliseconds. We generated the training data for this neural network with randomly chosen, uniformly distributed input parameters in ranges as defined in Table I. For this surrogate model, we used 546,689 samples created with ASTRA. We used min-max normalization as described in Eq. (2) for parameters and simulation output, where x is the input to normalize, and x' is the normalized output. The neural network consists of five layers (input layer excluded). The number of neurons increases to 2002 in the first layer, after that decreasing logarithmically to 447, 100, and 20 neurons, and finally returning five outputs. Experiments have shown that this architecture provides the smallest error. The overall mean squared error between the simulation and the trained surrogate model is about 1.13×10^{-5} [17]. It is important to note that one should only use the surrogate model within the ranges specified in Table I. The error can be substantial for parameters outside this range since neural networks cannot extrapolate their trained parameter range.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}. \tag{2}$$

Offset finding. Finding the offset means determining the difference between the parameters in the simulation and the

TABLE I. Parameters and their ranges for the ASTRA simulations. All values are chosen randomly from a uniform distribution of the specified interval. Fixed values: Bunch charge scale: 0.1 pC, solenoid position in respect to the cavity in z axis: 0.4625 m, stop position of tracking: 1.737 m longitudinal offset of the input distribution like gun peak field but multiplied with 10^{-4}. The state parameters are $s \coloneqq [s_1, \ldots, s_8]$, the action parameters $a \coloneqq [a_1, \ldots, a_4]$ and the integral parameters $t \coloneqq [t_1, t_2]$.

Label	Parameter	Interval	Unit
s_1	Laser pulse length	$[0.6, 4] \times 10^{-3}$	ns
s_2	Laser spot size on cathode	[0.2, 0.8]	mm
s_3	Gun peak field	[9, 18]	MV/m
s_4	Gun dc bias field	[3, 5]	kV
s_5	Cathode position	$[-20, -5]$	0.1 mm
s_6	Field flatness	$[-0.5, 0.5]$	
s_7	Laser horizontal position	$[-1.5, 1.5]$	mm
s_8	Laser vertical position	$[-1.5, 1.5]$	mm
a_1	Solenoid horizontal position	$[-4, 4] \times 10^{-3}$	mm
a_2	Solenoid vertical position	$[-4, 4] \times 10^{-3}$	mm
a_3	Solenoid angle y axis	$[-30, 30] \times 10^{-3}$	rad
a_4	Solenoid angle x axis	$[-30, 30] \times 10^{-3}$	rad
t_1	Emission phase	$[-10, 70]$	deg
t_2	Solenoid strength	$[-0.1, 0.1]$	T

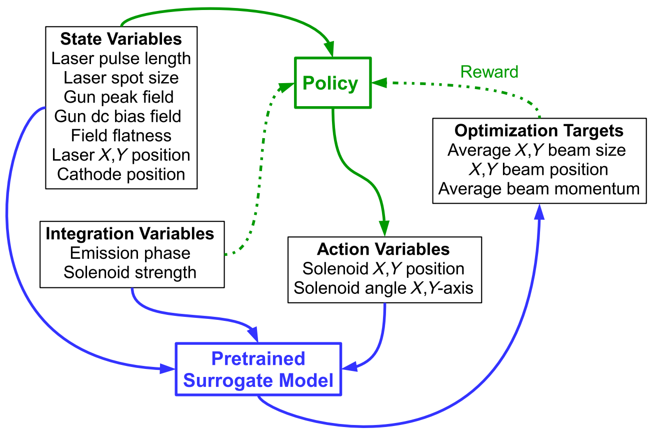

FIG. 4. Schema of the parameters' role within the learning loop. The blue arrows indicate the transitions within the pretrained surrogate model. The green elements indicate the training and decision-making by the RL agent. First, the RL agent determines how to choose the action variables using his policy. The state and integration variables are chosen randomly, but we can later measure them in the real experiment. Together with the state and integration variables, these actions get evaluated by the pretrained surrogate model. It returns the optimization targets, which serve as a reward for the policy, from which the RL agent can then improve its policy. Now the learning cycle repeats.

real device. Our method finds the offsets of the input parameters and output screen variables.

For solving the offset optimization problem, we used a local optimization method as proposed in [17]. However, local optimization algorithms rely on thousands of simulation evaluations to solve the optimization problem. That makes optimization computationally infeasible.

To find the offsets, we used the basinhopping algorithm [18]. Basinhopping works by walking a step randomly. After that step, it runs a local minimization algorithm. If the accuracy increases, it gets accepted and executes a new step. If the accuracy decreases, the algorithm discards the current step and randomly chooses another step [18].

For testing purposes, we assumed random offsets, which our algorithm needs to approximate. We add randomly generated offsets to the parameters as well as to the simulated output of these deviated parameters. These offsets are chosen from a uniform distribution with up to 20% deviation from the parameters' possible ranges as listed in Table I. By doing so, we receive a sample that imitates real device data with offsets. We also save the added random offsets since those serve as a reference for assessing the precision of the determined offset. We applied basinhopping with 200,000 iterations, a starting point $x_0 = 0$, stepsize 0.001, and a temperature for the acceptance criterion of 1.0. With a maximum deviation of 0.1, the basinhopping algorithm found the offsets with a sum-of-squares error of 8.25×10^{-5}. We assume that this level of accuracy is sufficient for the subsequent optimization steps. This assumption is based on the fact that the offsets could be checked independently from the input parameters via the output screen variables, as shown in [17].

Optimization. We will now take a closer look at the beam optimization task. We give a brief overview of the interaction of all parameters and optimization targets in Fig. 4.

The parameters of the electron gun (Table I) are divided into three groups: (i) *State parameters* s: We can only observe but not change these parameters. That includes the laser pulse length, the cathode's spot size, and the horizontal and vertical laser position. There are additional state parameters that describe the gun peak and bias field. The gun peak field is the maximal amplitude of the accelerating electrical field. Since the photocathode is electrically isolated from the rest of the gun cavity, an additional voltage can be applied, as described by the gun bias field. The field flatness characterizes the planarity of the cavity field, and another state parameter specifies the longitudinal cathode position. (ii) *Action parameters* a: Our agent can change these parameters. These are the solenoid horizontal and vertical positions and the angles with respect to the x and y axis. (iii) *Integral parameters* t: These parameters are like state parameters not modifiable by the agent but scanned over multiple equally distanced constant positions. An automated software procedure can easily change them in the real device. However, to limit our action space, we assume our agent cannot modify those, but the machine scans over them in a defined set of parameters. The two parameters in this group are the solenoid's focal strength and the electron gun's emission phase. The emission phase is the arrival time of the laser pulse relative to the sine wave of the high-frequency field of the electron gun.

The action parameters modified here are all enclosed in cryogenic encapsulation, which means they all have to be controlled by motors in the real device [19]. However, the electron gun of SeaLab is still not in commissioning yet. Therefore, we can only use simulated data for testing in this study.

Our optimization function is composed of four optimization criteria that will be named f_1 for the average horizontal beam size, f_2 for the average vertical beam size, and f_3 and f_4 for the horizontal and vertical beam position. The desired beam characteristics are a round and centered beam. In the ideal case, we reach $f_1 = f_2$ (round beam), $f_3 = 0$, and $f_4 = 0$ (centered beam). The parameters are denoted as state parameters $s \in S = [0,1]^8$, action parameters $a \in A = [0,1]^4$, and integral parameters $t \in T = [0, 0.9] \times [0.6, 0.9]$. We pick the state and action parameters randomly from their corresponding state space T and action space A. The integral parameters t are chosen evenly spaced from T, and we sample 20 data points per dimension. A detailed list of all parameters and their ranges is given in Table I. The optimization criteria are all functions $f_i : S \times A \times T \to \mathbb{R}$ for $i \in \{1,2,3,4\}$ and can be approximated using the proposed surrogate model from [17].

We define the optimization function, in the RL context often called the reward function, as follows:

$$\begin{aligned} R_1(s,a) &\coloneqq \sum_{t\in T} l[f_1(s,a,t) - f_2(s,a,t)] \\ R_2(s,a) &\coloneqq \sum_{t\in T} l[f_3(s,a,t)] \\ R_3(s,a) &\coloneqq \sum_{t\in T} l[f_4(s,a,t)] \\ R_l(s,a) &\coloneqq \min\left(\begin{bmatrix} R_1(s,a) \\ R_2(s,a) \\ R_3(s,a) \end{bmatrix}\right) \end{aligned} \tag{3}$$

The function $l(x) \coloneqq \min(-|x|, -\epsilon)$ limits the optimization of the components R_1, R_2, R_3 to a defined maximum accuracy level $\epsilon > 0$. That limitation leads to smoother convergence and avoids overfocusing on one component of the reward function. In the term of R_l, we choose the minimum function of the stacked component vector instead of the sum since it leads to faster convergence due to higher gradients. Please note that according to this definition, our reward is always nonpositive and values closer to zero are better.

Typically in RL problems, we would define a feedback loop considering step-based operations. However, in this case, this is not necessary because our environment does not require step-based actions and does not have delayed rewards. Furthermore, the states do not get modified during a learning cycle. Despite these specifications, we solve this problem with an RL model because the RL loop enables us to examine better why the agent chooses a particular action in a specific situation. Moreover, we can later easily replace the simulation with the real environment and perform further optimization after the initial learning phase with the simulated environment.

Since we consider the one-step case, we define the optimal policy we are looking for as

$$\mu^* \coloneqq \arg \max_{\mu} J(\mu) \tag{4}$$

with

$$J(\mu) \coloneqq \mathbb{E}_{s\sim p_0}[R_l(s, \mu(s))]. \tag{5}$$

A policy μ is a function that maps a state to action. The states s are chosen from some state distribution p_0.

According to the deterministic policy gradient theorem [20], we can assume

$$\nabla_\theta J(\mu_\theta) \approx \mathbb{E}_{s\sim p_0}[\nabla_\theta \mu_\theta(s) \nabla_a R_l(s,a)|_{a=\mu_\theta(s)}]. \tag{6}$$

We denote the policy with μ_θ since we will use a neural network as a policy that has parameters θ (weights of the connections between neurons) that we can learn through a training process. In our case, we can calculate $\nabla_a R_l(s,a)$ because our surrogate model is a neural network that we can differentiate (because a neural network is a composition of linearly combined nonlinear activation functions, which are differentiable, at least almost everywhere).

Our configuration for the maximum accuracy level is chosen as $\epsilon = 5 \times 10^{-5}$. We chose this value because this level of accuracy is adequate for the application. The state distribution p_0 is chosen from an eight-dimensional normal distribution:

$$p_0 \sim \mathcal{N}([0.5]^8, [0.04]^8). \tag{7}$$

The superscripted eight indicates the eight dimensions of the mean and variance. We chose the mean value $[0.5]^8$ to get centered samples since our data are normalized in the range [0, 1]. We do this cropping so that the agent cannot choose actions out of the allowed safe operation range. The variance is chosen as $[0.04]^8$ so that the resulting numbers are large enough to produce relevant states but not too large and thus rarely out of range. The values get truncated so that they stay within a range of [0, 1]. We do not require a Q-function because we only consider the one-timestep RL cycle, and the used surrogate model is differentiable. That allows us to determine the policy μ_θ with a policy gradient approach. We use a multilayer perceptron neural network with three hidden layers: 1000, 400, and 200 nodes, as depicted in Fig. 5. It uses four output nodes for the four action parameters. It is activated with the `ReLU` function

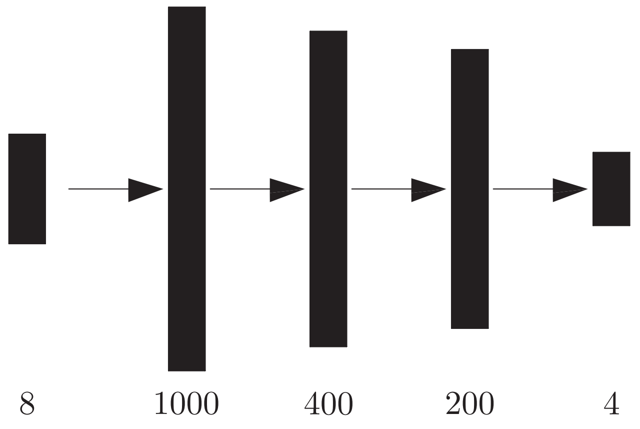

FIG. 5. Schematic plot of the neural network we use for learning the policy. We scaled the heights of the layers logarithmically for visualization purposes. The numbers indicate the layer sizes, input and output dimensions.

except for the output layer, which uses `Tanh` for activation. We use the optimizer `Adam` (adaptive moment estimation) [21] with a learning rate $\eta = 10^{-4}$. The learning rate specifies how much the error of new batches is weighted when updating the neural network weights. The training was batched, so the weights need to be updated less frequently. The chosen batch size was 32 samples per batch. We chose parameters similar to a comparable setting where the booster current parameters were optimized, as described in [8]. For better training stability, we use min-max normalization for the inputs and outputs of the surrogate model when training the policy. We performed a brief hyperparameter search with different amounts of layers and nodes. However, the hyperparameters similar to [8] achieved the largest rewards. We train our agent for more than 21,875 epochs. One epoch consists of training one batch, i.e., 32 samples, which means we perform the RL cycle and thus reward evaluations 700,000 times. This amount of repetitions is possible without escalating in time because we have a fast surrogate model. The calculation of the policy takes about 35 min on an Intel Xeon Gold 6252 processor. We depict the training process in Fig. 11. It is worth noting that the learning process saturates after about 20,000 epochs, which means that it improves only slightly after that. After this initial training phase, we freeze the policy. We can determine the chosen actions to arbitrary state and integration variables and calculate their reward with only a single surrogate model evaluation.

IV. RESULTS AND DISCUSSION

In this section, we present the results of our method and discuss them in regard to the research questions proposed in Sec. I.

A. Fast inference requiring fewer reward evaluations

We compare the optimization performance of the trained policy with four different optimization algorithms as baselines. The optimizers try to solve the following optimization problem:

$$\arg\min_{a} R_l(s,a). \tag{8}$$

The state s is sampled from p_0 as defined in Sec. III. We will compare the optimal value $R_l(s,a)$ after 1000 function evaluations with our RL approach $R_l[s,\mu(s)]$. We assume

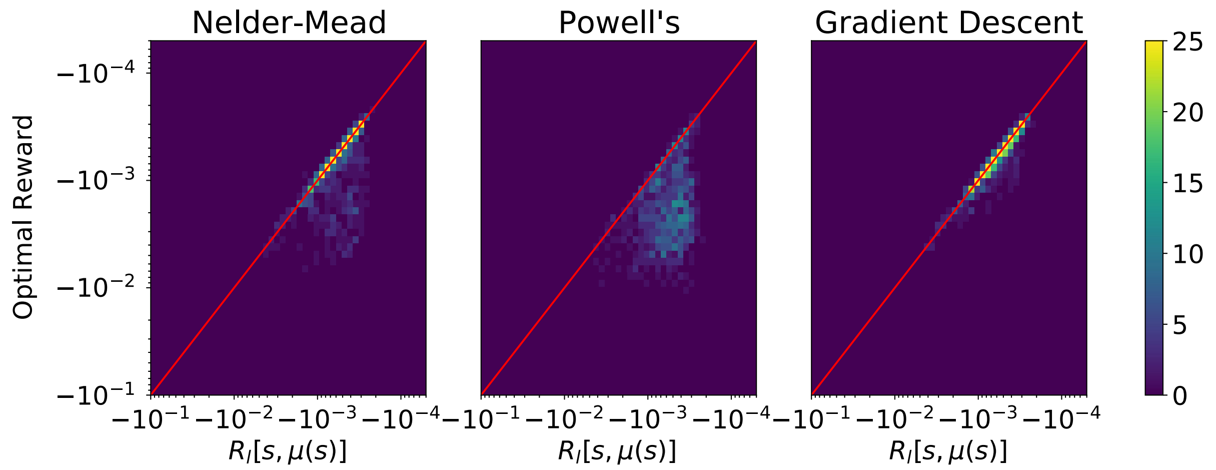

FIG. 6. Comparison of the evaluated RL policy $\mu(s)$ with the best reward achieved by the different optimization methods after 1000 reward function evaluations; Because the local optimization algorithms are stochastic, i.e., depend on random variables, we run the optimization algorithms 800 times to get a reliable amount of statistics. We plot the histograms with respect to the optimal rewards of our RL agent. The intensity shows how many runs of the stochastic optimizations have achieved a particular optimal reward. In case one method is performing equally as RL, only the bisecting line (highlighted in red) will be visible. If there is intensity below this line, the optimization method could not achieve a similar or better reward than the RL policy $\mu(s)$ within the allowed reward function evaluations. Note the logarithmic scale on both axes.

TABLE II. The second column shows the number of runs that achieved similar or better rewards than our RL approach with the corresponding method. The third column provides the number of reward function evaluations that were required until our RL approach was matched.

Method	Runs $\mu(s)$ was matched	Reward evaluations for matching $\mu(s)$
Nelder-Mead	457/800	230.53 ± 132.60
Powell's	57/800	499.26 ± 214.43
Gradient descent	396/800	586.44 ± 233.87

that the policy of our RL approach has been fully trained. The local optimization algorithms are stochastic, which means they use random variables for optimization. Therefore, we repeat the experiments 800 times to eliminate the possibility that the results are just coincidental. This approach allows us to calculate a mean value and avoids getting particularly good or bad results only due to coincidence. For Powell's and Nelder-Mead, we choose the default settings, which means that we set the absolute error in inputs and outputs between iterations that is acceptable for convergence to 0.0001. As a gradient descent algorithm, we use stochastic gradient descent with a learning rate $\nu = 0.1$. The histograms in the comparison plots in Fig. 6 have a higher value in the color scale below the red line. This shows that all other compared optimization methods reach a smaller reward given the same number of reward function evaluations. The results shown in Table II confirm these findings. All compared optimizers averagely fail to achieve equal or higher rewards than the RL approach. Only Nelder-Mead and gradient descent can achieve similar or better rewards than the RL policy in about half of the repetitions after 1000 reward evaluations. Powell's can even match only in 57 repetitions with the reward achieved by the RL agent. The best stochastic optimization method is Nelder-Mead, which can compete with the RL approach after 230.53 reward and thus surrogate model evaluations.

We will now compare the achieved rewards at different evaluation counts of the local optimization algorithms and our RL agent policy. With evaluation counts, we mean the number of evaluations of the reward function and, thus, the number of simulation approximations. We can equalize this amount as computational costs since evaluating the surrogate model takes the most time in the learning cycle of both the RL approach and the local optimization algorithms. We expect our RL policy to outperform the local optimization algorithms regarding the required evaluation counts for an adequate reward. In Fig. 7, we can see that even after 1000 evaluations, the RL policy has, on average, a larger reward than all compared local optimization algorithms. The reward of Powell's increases a lot slower and converges at a lower level than the RL policy. Both Nelder-Mead and gradient descent perform almost equally. However, even after 200 evaluations, the RL policy achieves a larger reward.

To summarize, we can conclude that our RL agent requires only one surrogate model evaluation compared to the local optimizers. This result means that our RL agent, once trained, provides considerably faster inference. The RL agent requires about 2.71 ms for choosing an action to a given parameter input combination on an Intel i5-7200U. We determined this value by averaging over 100,000 evaluations. This is also true for the approach shown in [15], which requires approximately ten surrogate model evaluations for the genetic algorithm. However, we would require an evaluation using the same surrogate model and optimization targets for a quantitative comparison of the prediction accuracy.

B. Compound solution for the optimization task

Together with the offset finding algorithm, we are now able to fully solve the beam optimization task. That means

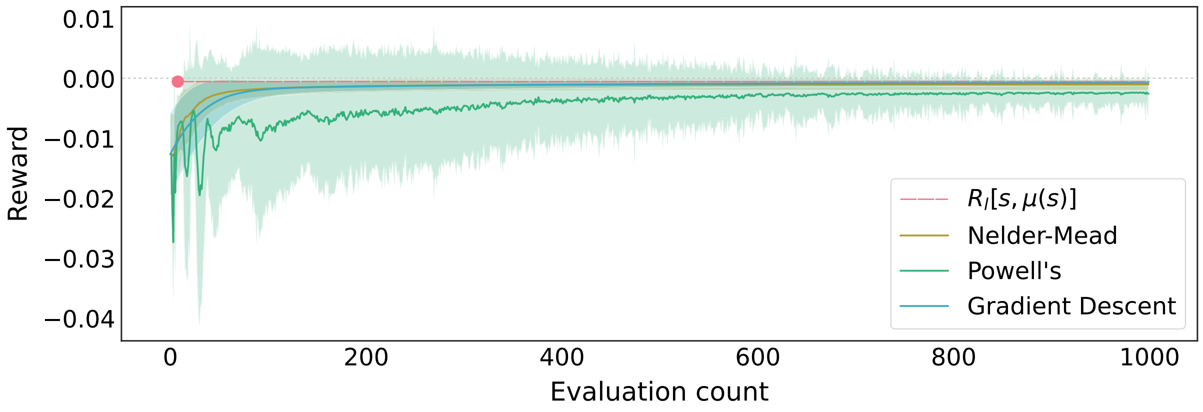

FIG. 7. Comparison of the evaluated RL policy $\mu(s)$ with local optimization algorithms with respect to achieved reward at different evaluation counts. A larger reward is better. We show the mean of 800 repetitions over 1000 evaluation steps. The standard deviation is plotted semitransparent. The RL policy is plotted as a dashed line to highlight that it is only a fixed value since, after completing training, it only requires one evaluation of the reward function.

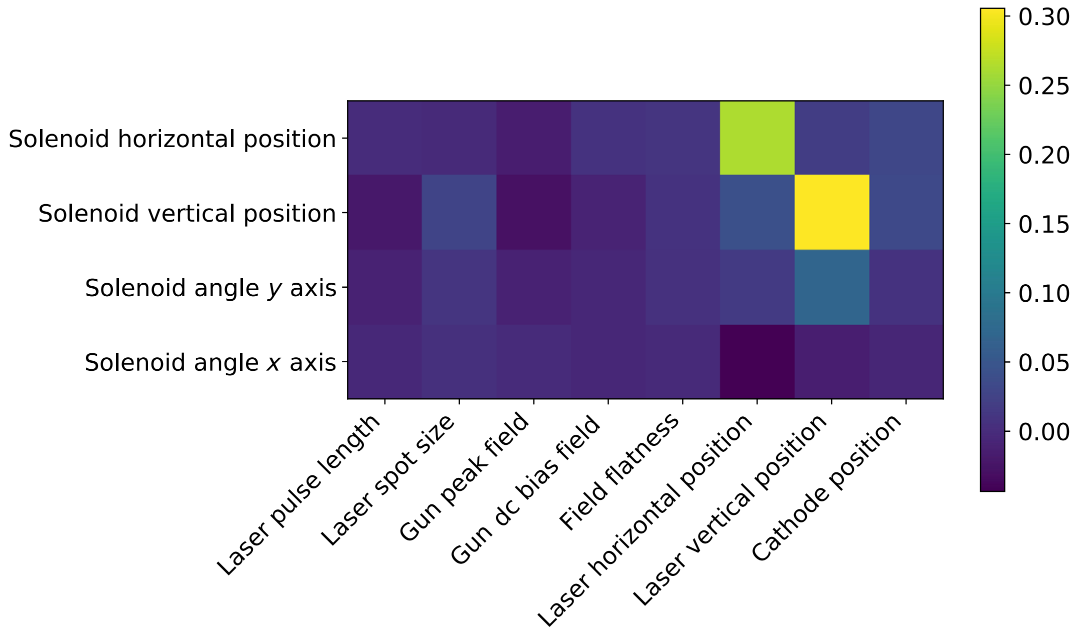

FIG. 8. Average of the Jacobian matrix of 100,000 state values and their chosen actions; A high value in the color scale means the average of the partial derivatives is high. It turns out that great changes in the laser horizontal position and vertical position lead to great changes in the respective solenoid position in the policy of our RL agent.

we need first to determine the offsets of the parameters and then apply the RL agent.

Our simulation method makes some assumptions, especially in the area of the electron source itself. We assume an exactly round beam at the beginning of the simulations and that the laser spot is homogeneous and stable in time.

In reality, it can happen that the laser spot on the cathode is not homogeneously round. Additionally, there are field errors in the electron gun, which we cannot estimate in all details yet. We cover these mainly in the variable of the field flatness. However, we do not consider higher asymmetries of the resonator (e.g., inner cell

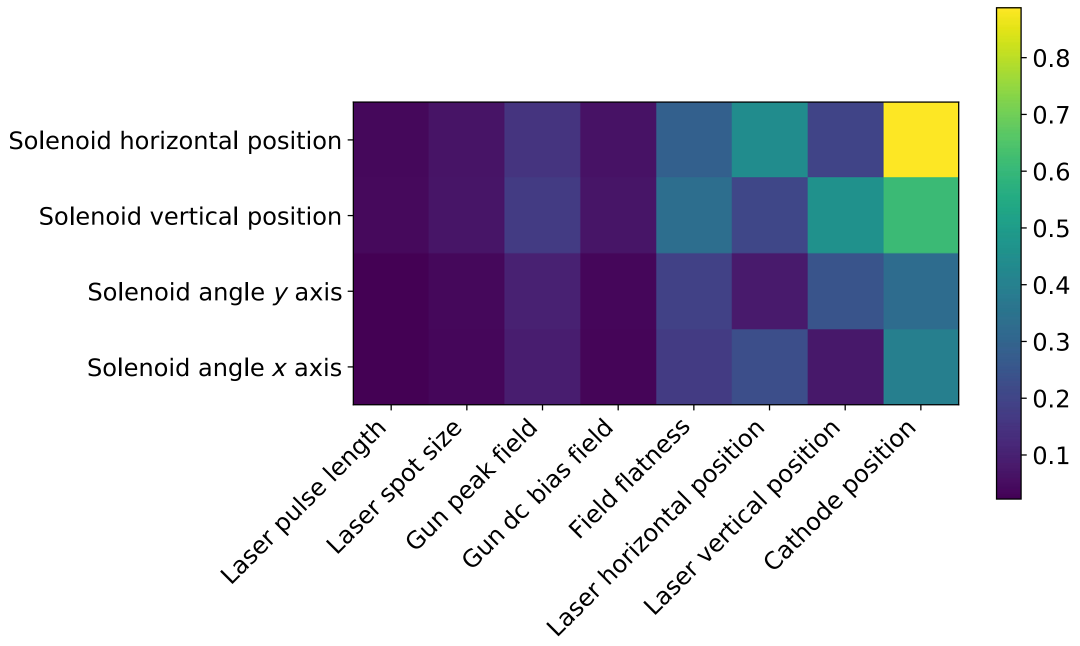

FIG. 9. Standard deviations of the Jacobian matrix of 100,000 state values and their chosen actions; A high value in the color scale means the standard deviation of the partial derivatives is high. It shows that the derivatives of the solenoid position have high fluctuations both vertically and horizontally with respect to the cathode position.

misalignment, coupler kicks, and higher rf modes). But these are higher-order effects and, therefore, should not significantly impact the applicability of our method in the real world.

The differences between the simulation and surrogate model are negligible within the trained parameter ranges. This premise allows us to safely use the surrogate model to replace the simulation in offset finding and beam optimization. In summary, based on our reasoning, our method solves the optimization task and is transferable to the real-world application.

C. Explainability of decisions

In the following, we will elaborate on how our RL agent chooses the actions according to a given state. We utilize the fact that we have trained a deterministic policy with a neural network. Since our neural network is differentiable (almost everywhere), we can extract the Jacobian matrix with the help of automatic differentiation methods. Because the Jacobian matrix shows the actions' derivatives concerning the different state values, we can see which action has a high impact on the respective state value. We depict the mean and standard deviation of the policy Jacobian matrices evaluated at 100,000 states sampled from a normal distribution in Figs. 8 and 9. Figure 8 shows that the average change of the action solenoid horizontal position is high when the horizontal laser position changes. This relation can be seen explicitly in Fig. 10, where we applied varied inputs to the policy of our RL approach. There is also a high correlation between the vertical laser position and the action parameter vertical solenoid position. Figure 9 indicates that there are high fluctuations in the derivatives of solenoid horizontal and vertical position concerning the cathode position. Since

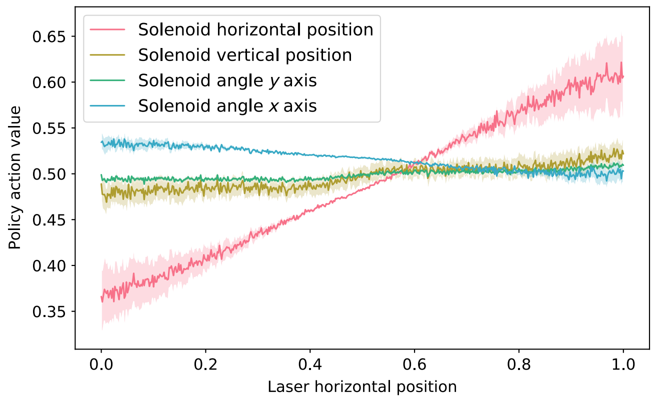

FIG. 10. Random action values ordered by laser horizontal positions; The standard deviation of the input values is $\sigma = 0.2$. This plot shows the mean of 500 laser horizontal positions and 1000 repetitions. The variance is plotted semitransparent. The plot shows that with increasing laser horizontal position, also the solenoid horizontal position is increased. The solenoid angle in the y axis is also moved slightly. However, the solenoid vertical position and solenoid angle x axis are almost not altered.

the mean of the Jacobian matrix is low at this point, the impact of the solenoid positions for aligning the cathode position is minimal. This examination of the policy visualizes the way the RL agent chooses its actions and makes the decisions of the RL algorithm more transparent and, thus, more trustworthy. We stress that our RL agent can, by design, only choose actions in allowed and safe operation ranges because of the tanh output layer in combination with normalized outputs.

V. CONCLUSION

As shown in this article, we have successfully applied RL to optimize an SRF cavity module in the simulation environment. The used surrogate model as a fast approximation for the simulation is accurate and can safely replace the simulation within the defined parameter ranges. We have shown that the optimization accuracy of a pretrained RL agent is comparable with several local optimization algorithms but achieves this performance by direct evaluation instead of several hundred iteration steps. After training the RL agent, the inference times are much lower since the optimization problem only needs to be evaluated once instead of many times as with the local optimizers. This result is a considerable step toward fast and automated commissioning and optimization of an SRF gun. Potentially, this procedure will replace the time-consuming manual alignment in the future. This approach allows a quicker and easier setup of energy recovery linacs and other high-current and high-repetition electron beam applications.

The next step will be to verify our results on the actual device when it is ready for commissioning. Our approach should be directly applicable since the simulation already models inaccuracies of the device as discussed in Sec. IV. Probably, we can transfer this approach to other problems. For example, we could use it for optimizing beamlines of synchrotron radiation facilities. Another idea is to use real feedback loops during operation for applying RL. This should be done after initial training and optimization with the simulation approximation. That means that instead of relying only on our simulation approximation, we could also measure the state and integration variables in the actual device and then perform further policy optimization. To realize this idea, we needed to extend our approach by step-based actions due to hardware constraints, e.g., the motors performing the actions require some time for movement.

Dataset generation and program scripts to this article can be found at a repository hosted at Gitlab of Helmholtz-Zentrum Berlin [22].

ACKNOWLEDGMENTS

Support by the JointLab Artificial Intelligence Methods for Experiment Design (AIM-ED) between Helmholtz-Zentrum für Materialien und Energie, Berlin and the University of Kassel is gratefully acknowledged.

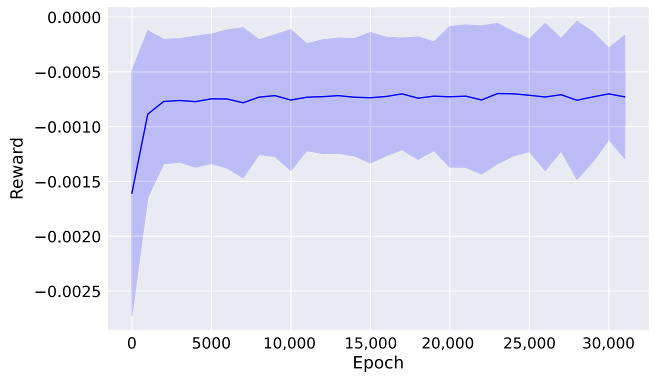

FIG. 11. The training process of the policy. Since the training is stochastic, we repeat it 500 times to get sufficient statistical evidence. The opaque graph shows the mean of all repetitions and the standard deviation is plotted semitransparent. The achieved reward improves only slightly after 20,000 epochs.

APPENDIX: POLICY TRAINING

We show the process of training the reinforcement agent's policy in Fig. 11. Please note that we need to train the policy only once, and then we can apply it to any input with a single surrogate evaluation.

[1] A. Neumann *et al.*, First commissioning of an SRF Photo-Injector Module for BERLinPro, in *Proceedings of 8th International Particle Accelerator Conference, IPAC-2017, Copenhagen, Denmark, 2017* (JACoW, Geneva, Switzerland, 2017), pp. 971–974.

[2] R. Sutton, R. Barto, A. Barto, C. Barto, F. Bach, and M. Press, *Reinforcement Learning: An Introduction*, A Bradford Book (MIT Press, Cambridge, MA, 1998).

[3] W. Lötzsch, J. Vitay, and F. Hamker, Training a deep policy gradient-based neural network with asynchronous learners on a simulated robotic problem, in *INFORMATIK 2017*, edited by M. Eibl and M. Gaedke (Gesellschaft für Informatik, Bonn, 2017), pp. 2143–2154.

[4] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Mach. Learn. **8**, 229 (1992).

[5] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, Deterministic policy gradient algorithms, in *Proceedings of International Conference on Machine Learning, ICML 2014, Beijing, China, 2014*, pp. 387–395, https://proceedings.mlr.press/v32/silver14.pdf.

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, Continuous control with deep reinforcement learning, in *Proceedings of 4th International Conference on Learning Representations, ICLR 2016*, edited by Y. Bengio and Y. LeCun (2016), https://arxiv.org/abs/1509.02971.

[7] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, An introduction to deep reinforcement learning, Found. Trends Mach. Learn. **11**, 219 (2018).

[8] L. V. Ramirez, G. Hartmann, T. Mertens, R. Müller, and J. Viefhaus, in *Proceedings of International Conference on Accelerator and Large Experimental Physics Control Systems, ICALEPCS 2019* (JACoW Publishing, Geneva, Switzerland, 2020), pp. 754–760.

[9] F. Gao and L. Han, Implementing the Nelder-Mead simplex algorithm with adaptive parameters, Comput. Optim. Appl. **51**, 259 (2012).

[10] M. J. D. Powell, An efficient method for finding the minimum of a function of several variables without calculating derivatives, Comput. J. **7**, 155 (1964).

[11] S. Ruder, An overview of gradient descent optimization algorithms, arXiv:1609.04747.

[12] J. Zhu, Y. Chen, F. Brinker, W. Decking, S. Tomin, and H. Schlarb, High-Fidelity Prediction of Megapixel Longitudinal Phase-Space Images of Electron Beams Using Encoder-Decoder Neural Networks, Phys. Rev. Appl. **16**, 024005 (2021).

[13] R. Roussel, A. Hanuka, and A. Edelen, Multiobjective Bayesian optimization for online accelerator tuning, Phys. Rev. Accel. Beams **24**, 062801 (2021).

[14] A. Edelen, N. Neveu, M. Frey, Y. Huber, C. Mayes, and A. Adelmann, Machine learning for orders of magnitude speedup in multiobjective optimization of particle accelerator systems, Phys. Rev. Accel. Beams **23**, 044601 (2020).

[15] R. Bellotti, R. Boiger, and A. Adelmann, Fast, efficient and flexible particle accelerator optimisation using densely connected and invertible neural networks, Information **12**, 351 (2021).

[16] K. Flöttmann, Astra—a space charge tracking algorithm version 3.2, DESY, Hamburg, Germany, 2017.

[17] D. Meier, G. Hartmann, J. Völker, J. Viefhaus, and B. Sick, Reconstruction of offsets of an electron gun using deep learning and an optimization algorithm, in *Proceedings of Advances in Computational Methods for X-Ray Optics V*, edited by O. Chubar and K. Sawhney (International Society for Optics and Photonics, SPIE, Bellingham, WA, 2020), pp. 71–77.

[18] D. Wales and J. Doye, Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms, J. Phys. Chem. A **101**, 5111 (1997).

[19] G. Kourkafas, A. Jankowiak, T. Kamps, J. Li, M. Schebek, J. Völker *et al.*, Solenoid Alignment for the SRF Photo-injector of BERLinPro at HZB, in *Proceedings of 8th International Particle Accelerator Conference, IPAC 2017* (JACoW, Geneva, Switzerland, 2017), pp. 1778–1780.

[20] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in *Proceedings of Advances in Neural Information Processing Systems 12, NIPS 1999* (MIT Press, Cambridge, MA, USA, 1999), pp. 1057–1063.

[21] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, in *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA* (2015), https://arxiv.org/abs/1412.6980.

[22] https://gitlab.helmholtz-berlin.de/sealab/rl-optimization