Emukit: A Python toolkit for decision making under uncertainty

Andrei Paleyes[‡*], Maren Mahsereci[§], Neil D. Lawrence[‡]

Abstract—Emukit is a highly flexible Python toolkit for enriching decision making under uncertainty with statistical emulation. It is particularly pertinent to complex processes and simulations where data are scarce or difficult to acquire. Emukit provides a common framework for a range of iterative methods that propagate well-calibrated uncertainty estimates within a design loop, such as Bayesian optimisation, Bayesian quadrature and experimental design. It also provides multi-fidelity modelling capabilities. We describe the software design of the package, illustrate usage of the main APIs, and showcase the breadth of use cases in which the library already has been used by the research community.

Index Terms—statistical emulation, software, Bayesian optimisation, Bayesian quadrature, Bayesian experimental design, multi-fidelity, active learning

Introduction

Data selection is a major challenge in supervised machine learning (ML). Quite often when data availability is not an issue, data collection occurs prior to the training process and results in a static dataset, meaning that the machine learning model has no influence on the data collection process. However, if data points are expensive and scarce the performance of a model trained on a static dataset can be suboptimal or even poor. In those cases, it is beneficial to carefully select the dataset such that, for example, it is maximally informative under the ML model to achieve the task at hand. This branch of ML is generally referred to as *active learning* [1] and has attracted attention in various sub-fields such as *experimental design* (the task of predicting an unknown function value from its input), *global optimisation* (guessing the global minimiser of a function) and *integration* (guessing the integral of a function). The Emukit Python library, at core, augments existing machine learning models with active data selection functionality.

Tasks where data acquisition is hard usually involve a higher degree of expert knowledge on the modeling side, because incorporating prior information, such as mechanical or physical knowledge about the system under study, aims for more physically meaningful and accurate predictions on the task at hand. Often these models are of probabilistic nature and can provide a degree of uncertainty of their prediction to counteract the lack of data [2], [3], [4], [5]. Such a model is often referred to as a *statistical emulator*[1], which is a machine learning model that can replace an expensive computer simulation (a *simulator*) or real world experiment, and is trained on input-output pairs of the latter [6], [7], [8], [9]. Concretely, the simulator could be an involved stochastic weather simulation, and the emulator a predictive machine learning model trained on expensive input-output pairs of the weather simulation. Alternatively, the emulator may model a real world process. The emulator now may replace the original data source to obtain fast predictions when needed, or to compute auxiliary quantities that cannot be obtained from the data source.

Once trained, the performance of the emulator (the ML model), as mentioned, depends on the informativity of the data (the simulation results), especially if it is expensive and scarce, and hence active data selection is often desirable for such models. A unique feature of Emukit is that it enables the user to wrap custom emulator models into an interface provided by Emukit and, by doing so, use them in Emukit's decision loop. As such, Emukit 'actifies' (makes active) the data-acquisition of custom models written in custom backends that only connect via an interface to Emukit. This may i) save users time and money to write their own active learning loop, ii) or to rewrite their custom model in existing decision loop packages with a fixed backend, and iii) improve performance of the model with more informative training data.

Hence, the most prominent features of Emukit can be summarized as follows.

- Emukit *augments existing models with active learning* capability, in particular models used in Bayesian optimisation, Bayesian quadrature and experimental design.
- Emukit can use existing, potentially specialized, custom models provided by the user and wrap them into a provided interface. As such Emukit is *model backend agnostic*.
- Emukit is highly abstracted and mimics the components of an active decision loop. This composability allows users to provide custom implementations of subroutines and classes that seamlessly integrate with the rest of the package. Hence Emukit is *highly flexible* and allows *fast and easy prototyping*.
- In contrast to other packages, Emukit provides several active learning methods via *subpackages that share a core*

* Corresponding author: ap2169@cam.ac.uk
‡ Department of Computer Science and Technology, University of Cambridge
§ University of Tübingen

1. The reader might be familiar with the term 'emulator' in computing context, where it refers to a hardware or software that makes one system behave like another. The '(statistical) emulator' we use throughout this paper is an unrelated, albeit similar, term from the machine learning literature.

implementation of the active learning loop. This enables the user to potentially use the same model backend and even the same model instance across tasks. This increases consistency between results, may reduce implementation overhead and allow resource sharing between tasks.

- Emukit provides basic functionality for *multi-fidelity modeling* which allows the user to incorporate data sources of different fidelities. Further, Emukit contains a limited number of model wrappers to illustrate their usage and some example applications.

The remainder of the paper corroborates the points above in greater detail. The following section briefly introduces the supported machine learning methods before sketching Emukit's workflow and library structure. Throughout the text, we refer to 'tasks' in an abstract sense, without a specific application in mind. In the remainder of text, we will use the terms 'ML model' and 'emulator' interchangeably. The 'simulator' or 'data source' will later also be referred to as 'user function', 'black-box function' or 'objective function'.

Background on probabilistic active methods

This section gives an overview of the machine learning methods provided by Emukit. Emukit mainly contains three high-level methods: Bayesian optimisation (BO), Bayesian quadrature (BQ) and experimental design (ED).

Bayesian optimisation [10], [11] is a numerical method that aims to guess the global minimiser of a black-box function by querying function values at nodes and returning the minimiser of the collected set. Corresponding algorithms are inherently sequential and, at every iteration, decide on where to query the objective function next. The decision solves the so called 'exploration-exploitation trade-off' between exploring unknown regions of the function's domain, or exploiting rather promising regions of a potential minimiser. This trade-off is encoded in a heuristic called 'acquisition function' that quantifies the usefulness of evaluating the function at a certain node. Hence, BO is generally sample-efficient and thus especially useful when the function is expensive to evaluate and the number of allowed evaluations is limited. There exists a large range of heuristics and methods that all fall under the umbrella of BO, out of which Emukit supports several. Bayesian optimisation has been successfully applied in various fields [12], [13], but most notably in the automation of hyperparameter tuning tasks of neural networks [14], [15].

Bayesian quadrature [16], [17], [18] is a numerical method that aims to infer the integral of a black-box function (called the 'integrand') given some integration measure and queries of the integrand at nodes. In contrast to Monte Carlo (MC) methods, BQ generally accept any kind of node design and is especially sample efficient which makes it superior to MC in certain circumstances [19]. A sub-group of BQ methods are active and follow a similar decision loop as BO; the most notable difference being that acquisition functions are specific to BQ and the class of models is somewhat more restricted. Generally, active BQ methods are algorithmically similar on a high level to BO methods and can use similar models.

Experimental design [6], [7], [8], [9], also known as Bayesian active learning, is a method that aims to collect data about a black-box function such that the resulting probabilistic model predicts unseen function values well. Unlike BO and BQ discussed above, ED aims to learn the objective function as well as possible across the entire input space. It traditionally has been applied to statistical emulation of complex computer models but also has found applications in healthcare [20], computational biology [21] and engineering [22]. Some ED methods also obey the structure of an active decision loop similar to BO and BQ.

Emukit embodies the realization that all three methods (BO, BQ, ED), albeit having different numerical aims, share the same algorithmic structure: a decision loop that computes the next node based on the current model, evaluates the user function, and then updates the model accordingly. This decision loop is contained in Emukit's core package and shared by all three high-level methods. Furthermore, especially BO and ED may use similar models while BQ is somewhat more restricted. Potential benefits of sharing implementation, model and compute between tasks are discussed in later sections.

Finally, Emukit provides basic support of *multi-fidelity models* [23] which can combine query results of the black-box function of different quality (from low fidelity to high fidelity). This yields multi-fidelity BO, BQ and ED methods that may be made active again with Emukit's decision loop.

Emukit workflow

Decision making with statistical emulation consists of three parts. All starts with a *task*, a high level goal that we are interested in achieving. It usually involves a complex process that we aim to study to answer a question. Some examples include finding the best operation mode of a drone, measuring the quality of a weather simulation, explaining behavior of a complex system. In order to solve the task we choose a *method*, a relatively low-level technique that guides our exploration of the target process and provides the quantifiable way to answer the task's question. Examples include Bayesian optimisation, Bayesian quadrature and experimental design. And finally there is a *model*, a probabilistic data-driven representation of the process under study. Examples of such models are a Gaussian process, a random forest or a Bayesian network. Consequently, the typical workflow for users working with Emukit consists of three steps (see Figure 1 for a graphical description).

Build the model. Instead of constraining the user to certain model classes, Emukit provides the flexibility of using user-specified models. Generally speaking, Emukit does not provide modeling capabilities. Instead users are expected to define their own models. Because of the variety of modeling frameworks available, Emukit does not mandate or make any assumptions about a particular modeling technique or a library, and suggests implementing a subset of defined model interfaces that are required to use a particular method.

Run the method. This is the main focus of Emukit. Emukit defines a general structure of a decision making method and offers implementations of several such methods: Bayesian optimisation, Bayesian quadrature, experimental design. All methods are model-agnostic and only rely on model interfaces.

Solve the task. For the end users, Emukit is a way to solve a certain task, which may have research or business value. Emukit provides a set of examples of how tasks such as hyperparameter tuning, sensitivity analysis, multi-fidelity modeling or benchmarking are accomplished using the library.

Structure of the library

At a conceptual level the methods supported in Emukit – such as Bayesian optimisation, experimental design and Bayesian quadra-

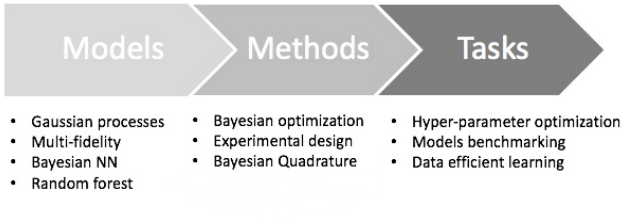

Fig. 1: *Summary of workflow for the users of Emukit. The user chooses a a modeling framework and defines a model. The model is wrapped using a pre-defined interface and connected to the core components of several methods such as Bayesian optimisation, experimental design etc. Specific tasks are then solved using these methods.*

ture – are all iterative decision making processes that follow a similar pattern. Algorithmically they can be thought of as instances of a common abstract loop, which we now describe (also see Algorithm 1).

The common goal of all of these methods is to learn a behavior of an *objective function* - a black-box expensive process that has certain *parameters*. The knowledge about the objective function (initially available as well as that collected during the learning process) is represented with a *probabilistic model*. New data points are proposed by optimising an *acquisition function* constructed using the model. Finally, the decision making process is done in a *loop* until a certain *stopping condition* is met.

Algorithm 1 Decision making loop in Emukit.

1: **while** stopping condition is not met **do**
2: collect next point(s) for evaluation
3: evaluate objective function
4: update model with new observation(s)
5: **end while**

The internal structure of Emukit reflects these abstractions to enable swapping and replacement of fundamental components of the decision making loop. While some of the basic components in Emukit correspond to the parts of the decision making loop exactly, others are more fine-grained to allow for greater flexibility and plug-and-play experience for the researchers using the package. We will now give an overview of these components.

Outer Loop. The `OuterLoop` class is the abstract loop where the different components come together. Loops for specific methods, such as Bayesian optimisation and experiment design, should subclass it. The library provides several concrete implementations of the loop, and also contains examples how the users may build their own.

Parameter space. Represents the parameter space of the objective function, also referred to as input space. Emukit supports continuous, categorical, discrete, and bandit parameters.

Model. All Emukit loops need a probabilistic model. Emukit does not provide functionality to build models as there are already good modeling frameworks available in Python. Instead, it provides a way of interfacing third-party modeling libraries. The interfacing mechanism consists of two parts: interfaces and wrappers. *Interfaces* define functionality required from a model. Different models and modeling frameworks will provide different functionality. For instance a Gaussian process will usually have derivatives of the predictions available but random forests will not. A model implements a set of interfaces that represent these different functionalities. The basic interface that all models must implement is `IModel`, which implements functionality to make predictions and update the model but a model may implement any number of other interfaces such as `IDifferentiable` which indicates a model has prediction derivatives available. Other components of the decision making loop may also be define interfaces to indicate that they require a certain functionality from the model. For example, `ICalculateVarianceReduction` defines methods the user needs to implement with their model to use it with the variance reduction technique. `Model wrappers` adapt third-party models and implement one or more of the interfaces using specific modeling framework. Emukit provides a wrapper for using a model created with `GPy` [24].

Candidate Point Calculator. This entity drives the decision on which point(s) to evaluate next. The simplest implementation provided out of the box, `SequentialPointCalculator`, collects one point at a time by finding where the acquisition is at a maximum by applying the acquisition optimiser to the acquisition function. More complex implementations are possible, for example to enable batches of points to be collected so that the user function may be evaluated in parallel.

Acquisition. The acquisition is a function defined on the parameter space that produces continuous values. It represents a heuristic quantification of how valuable collecting a future point might be, and produces continuous values. It is used by the candidate point calculator to decide which point(s) to collect next. Acquisition functions balance exploration and exploitation of the decision making process.

Acquisition Optimiser. The `AcquisitionOptimizer` optimises the acquisition function to find the point at which the acquisition is at a maximum. If available, the optimiser can use the acquisition function gradients. Otherwise, it will either estimate the gradients numerically or use a gradient free optimisation.

User Function. This is the component that represents the objective function. It can be evaluated by the user or it can be passed into the loop and evaluated by Emukit.

Model Updater. The `ModelUpdater` class updates the model with new training data after a new point is observed and optimises any hyperparameters of the model. It can decide whether hyperparameters need updating based on some internal logic.

Stopping Condition. The `StoppingCondition` class chooses when the decision making loop should stop collecting points. The most commonly used approach is to stop when a set number of iterations has been reached.

These are the core components Emukit defines. Specific methods may also define additional concepts of their own, e.g. integration measures or costs. Table 1 shows the mapping between decision making abstractions and Emukit components.

USAGE OVERVIEW

This section describes Emukit's high level APIs for all main functions of the package: Bayesian optimisation, Bayesian quadrature, experimental design and multi-fidelity emulation. Unless stated otherwise, we assume that some initial data (an initial design of reasonable size with corresponding evaluations of the user function) are already defined and stored in the variables `X` (inputs) and `Y` (values). We use GPy [24] in the code snippets below for modeling, and exclude import lines for brevity.

Decision making abstractions	Emukit components
Loop	Outer loop
Parameters	Parameter space
Probabilistic model	Model interface
	Model wrapper
Acquisition function	Candidate point calculator
	Acquisition
	Acquisition optimiser
Objective function	User function
	Model updater
Stopping Condition	Stopping condition

TABLE 1: *The mapping between abstractions of the decision making process and the components defined in Emukit.*

Standard methods and model wrapping

Interfaces for Bayesian optimisation and experimental design are the most straighforward ways to use the library. Both methods require the user to define a model and wrap it in the Emukit's model wrapper. An input space also has to be defined using Emukit's classes. The choice of acquisition function is optional, as reasonable defaults are provided. High level loop objects allow the user to execute the decision making loop and access its properties.

```
model_gpy = GPy.models.GPRegression(X,Y)
model_emukit = GPyModelWrapper(model_gpy)

parameter_space = ParameterSpace([
    ContinuousParameter('x1', -5, 10),
    ContinuousParameter('x2', 0, 15)
])

expected_improvement_acquisition =
    ExpectedImprovement(model = model_emukit)
bayesopt_loop = BayesianOptimizationLoop(
    model = model_emukit,
    space = parameter_space,
    acquisition = expected_improvement_acquisition
)

model_variance_acquisition =
    ModelVariance(model = model_emukit)
experimental_design_loop =
    ExperimentalDesignLoop(
        model = model_emukit,
        space = parameter_space,
        acquisition = model_variance_acquisition
    )
```

Usage of Bayesian quadrature (BQ) API is more involved, as even in its most basic form it requires more choices from the user. First the objective function, also referred to as an integrand, is modeled with a Gaussian process (GP). Since BQ integrates the kernel function, the kernel is then wrapped in a separate Emukit object. Bundled together, wrappers around the kernel and the model itself represent a base model in the BQ package. This model may be used with several BQ methods, the code below illustrates vanilla Bayesian quadrature where the GP model is directly placed over the integrand function and then integrated analytically.

```
lb = -3.0 # lower integral bound
ub = 3.0 # upper integral bound
gpy_model = GPy.models.GPRegression(X=X, Y=Y)

emukit_rbf = RBFGPy(gpy_model.kern)
emukit_measure = LebesgueMeasure.from_bounds(
    bounds=[(lb, ub)]
)
emukit_qrbf = QuadratureRBFLebesgueMeasure(
    emukit_rbf, emukit_measure
)
gp_model = BaseGaussianProcessGPy(
    kern=emukit_qrbf, gpy_model=gpy_model
)

emukit_model = VanillaBayesianQuadrature(
    base_gp=gp_model, X=X, Y=Y
)
bq_loop = VanillaBayesianQuadratureLoop(
    model=emukit_model
)
```

Once the loop object is created, either for optimisation, quadrature or experiment design, it may be evaluated in one of two modes. If the user has access to the objective function via Python, Emukit can manage the loop with the `run_loop` method that accepts two arguments: the objective function and the stopping criterion. If the objective has to be called externally (e.g. a lab experiment has to be done), Emukit provides `get_next_points` method that produces the next evaluation point(s) based on the data observed so far. In that latter case user has to manage the decision making loop themselves.

Interfaces for fast prototyping

Emukit gives researchers a lot of flexibility in swapping individual pieces in and out of the decision making loop. This is made possible by clearly defined interfaces. We illustrate how this is accomplished in the package with an example of `IntegratedHyperParameterAcquisition`. This class provides an ability to integrate any acquisition function over hyperparameters of the model. To do that, the model needs to support two operations: generate hyperparameter samples and fix hyperparameters to a certain sample value. Consequently, Emukit defines an interface `IPriorHyperparameters` that declares these operations, and `IntegratedHyperParameterAcquisition` requires input model to implement this interfaces, as is shown in the following code snippet:

```
class IPriorHyperparameters:
    def generate_hyperparameters_samples(...

    def fix_model_hyperparameters(...


class IntegratedHyperParameterAcquisition(Acquisition):
    def __init__(
        self,
        model: Union[IModel, IPriorHyperparameters],
        ...
```

Model reuse across tasks

Emukit's composability allows to reuse components between methods. For example, we use the quadrature model defined above to perform an optimisation loop, and then integrate it using the quadrature API. The ability to reuse components in this way lowers implementation overhead, optimises utilisation of compute resources, and increases consistency.

```
# see BQ snippet for complete
# definition of the model
emukit_bq_model = VanillaBayesianQuadrature(
    base_gp=gp_model, X=X, Y=Y
)
```

```
bayesopt_loop = BayesianOptimizationLoop(
    model = emukit_bq_model, space = parameter_space
)
n_iterations = 20
bayesopt_loop.run_loop(
    user_function,
    stopping_condition=n_iterations
)

emukit_bq_model.integrate()
```

Multi-fidelity emulation

To support research on multi-fidelity emulation methods, Emukit implements both linear and non-linear multi-fidelity models. The user needs to provide data for each of the fidelities and make the choice of appropriate Gaussian process kernel. Emukit can then be used to define a combined multi-fidelity model. In the example below we define a linear multi-fidelity model, where the relationship between fidelities is linear.

```
# This utility method allows conversion
# of data from different fidelities
# to arrays where fidelity is represented
# as an input variable
X, Y = convert_xy_lists_to_arrays(
    [x_low, x_high],
    [y_low, y_high]
)

kernels = [
    GPy.kern.RBF(dim=1),
    GPy.kern.RBF(dim=1)
]
linear_mf_kernel =
    LinearMultiFidelityKernel(kernels)
gpy_linear_mf_model =
    GPyLinearMultiFidelityModel(
        X, Y,
        linear_mf_kernel,
        n_fidelities = 2
)
```

Other methods and features

In addition to the APIs discussed above, Emukit also provides basic support for sensitivity analysis and benchmarking. Further information about Emukit's functionality, including available implementations of acquisition functions, multi-output models, support for constraints and cost functions, and custom events in the outer loop may be found in library's website[2], documentation[3] and tutorial notebooks[4].

EMUKIT IN ACTION

Since its announcement in 2019 [25], Emukit was used in a wide range of research projects. In this section we review a selection of these projects to showcase the breadth of situations in which the library may be useful.

Methodological research

Because of its flexibility Emukit allows researchers to rapidly experiment with decision making methods in its suite. In this section we discuss several research papers that leverage this advantage to advance the field of decision making under uncertainty.

Optimisation of parameters in high dimensional structured data spaces is an increasingly important and challenging task. A common pattern is to use unsupervised learning methods to project parameters into low dimensional continuous representations, also known as latent spaces. There are multiple ways to approach the design of the Bayesian optimisation procedure on such latent spaces. Siivola et al. [26] studied the effects of various design choices. Namely, the effects of the dimensionality of the latent space, the optimisation bounds, and the choice of acquisition function were analysed. Emukit's plug-and-play approach allowed the researchers to facilitate measurement of these effects in isolation.

Emukit's composability was also leveraged for the implementation of BOSH, a sampling approach for Bayesian optimisation of functions with stochastic evaluations [27]. Authors used hierarchical Gaussian process as a surrogate and designed a novel BOSH acquisition function using the information-theoretic framework, incorporating both pieces in Emukit's Bayesian optimisation loop. Emukit was also used to assess BOSH performance against a variety of baselines.

Naslidnyk et al. [28] implemented a custom Bayesian quadrature model and used Emukit's existing BQ wrapper and decision loop in order to learn integrals of functions that are input invariant under some transformations. They tested their method on a problem from Fourier optics where the integral over a point spread functions of symmetric lenses was computed. Further, Gessner et al. [29] applied Emukit in the context of active multi-source Bayesian quadrature. The authors implemented a custom multi-source BQ model, a corresponding wrapper and even a custom multi-source acquisition function and point calculator which was possible due to Emukit's abstraction and plug-and-play capability.

Example applications

In this section we describe several cases where Emukit was used to solve applied research problems.

Bell et al. used Emukit to show how to conduct multiverse analysis for machine learning experiments [30]. Multiverse analysis was originally introduced in psychology, and allows researchers to explore the robustness and generality of claims by systematically examining the impact of different choices and variations in the experimental setup. The authors argue that the same concept can be applied to the machine learning: if a new technique, e.g. batch normalization, is proposed for an ML model, it should remain effective regardless of the model architecture, optimisation method, dataset, evaluation metric, and so on. The set of these variations comprises a multiverse, and needs to be explored effectively. The authors use surrogate modeling and Bayesian experimental design to systematically explore the effect of each choice. Emukit was chosen as an implementation tool because of the experimental design API it provides.

Uhrenholt and Jensen used Emukit's Bayesian optimisation module to solve the problem of finding settings of a musical synthesizer to produce a given sound [31]. A musical synthesizer produces sound by generating waveforms via oscillators. Created audio streams are then routed through a pipeline that consists (not necessary all) of mixing of separate streams, filtering, adding of noise, and saturation. Musicians can control the output sound by changing the configuration of the pipeline. In order to estimate the discrepancy between the produced sound and the target, the

2. https://emukit.github.io/
3. https://emukit.readthedocs.io/en/latest/
4. https://nbviewer.org/github/emukit/emukit/blob/main/notebooks/index.ipynb

authors designed a novel modeling approach, in which Gaussian process is used to model the distribution of the output's L2 norm. The flexibility of Emukit allowed to implement this customization directly, without necessary effort duplication. Emukit's API also facilitated a fair comparison to the standard Bayesian optimisation used as a baseline.

Liyanage et al. faced the problem of combining data from multiple particle accelerators, including Large Hadron Collider and Relativistic Heavy Ion Collider, to study the properties of quark-gluon plasma [32]. Nuclear collision experiments generate a large body of measurements with varying levels of uncertainty that would be expensive to quantify with simulations. Instead the authors proposed to use inexpensive statistical emulators and use transfer learning to leverage similarities between different heavy ion collisions systems. This new technique is based on multi-fidelity emulation, making Emukit an obvious implementation choice.

Related work

The Python ecosystem is rich with powerful scientific packages, including those for decision making methods.

In particular, *Bayesian optimisation* enjoys a wide selection of tools and frameworks. Spearmint [14] and GPyOpt [33] are among the first Python packages for Bayesian optimisation, the latter being an inspiration for the first release of Emukit. BoTorch [34] is a popular library for Bayesian optimisation based on PyTorch. Similarly, Trieste [35] also focuses on Bayesian optimisation but uses Tensorflow as a backend. More options, such as pyGPGO [36], scikit-optimize [37], RoBO [38], are also available.

For *Bayesian quadrature* and *Bayesian experimental design* the choice of frameworks is more scarce. Namely, bayesquad [39] appears to be another Python package for Bayesian quadrature. The Python library ProbNum [40] supports a variety of Bayesian quadrature methods, but it's lack of hyperparameter tuning capability reduces its practical relevance significantly in its current form. Optbayesexpt [41] and NEXTorch [42] provide Bayesian experimental design functionality adopted for their respective fields. Elements of experimental design can also be found in Trieste [35].

The key difference between Emukit and the mentioned libraries is the fact that Emukit does not dictate a particular modeling framework, allowing for flexibility in the choice of computational backends. In addition, Emukit does not focus on a single method and provides a common core set of abstractions for optimisation, quadrature and experimental design. Emukit provides a unique way of using the same model backend for all tasks, which increases consistency, reduces implementation and computing overheads.

Likewise, we were not able to locate a Python library other than Emukit that provides multi-fidelity emulation functionality. A notable package for research on multi-fidelity methods is MF2 [43] that implements a variety of multi-fidelity benchmark functions, but does not have modeling capabilities.

Looking at a wider family of optimisation libraries in Python, Optuna [44] is a popular choice for hyperparameter optimisation. Similarly to Emukit, Optuna is framework agnostic, however it provides a different set of optimisation methods, focusing on evolutionary, genetic and Monte Carlo based approaches. Finally, Ray Tune [45] is a well known scalable platform in Python on which other model optimisation frameworks can be executed. Emukit can potentially be integrated with Ray Tune as an optimisation library. This work was not carried out yet, and may be a future development direction.

Limitations

The design choices made in Emukit have proven to be highly beneficial for rapid prototyping and experimentation. However they also led to some of the key limitations of the library.

Emukit does not provide modeling capabilities, and instead requires users to provide their own surrogate models. This requires certain level of proficiency with probabilistic modeling, and can prevent some people from using the library. While we aim to mitigate this with extensive collection of examples and tutorials, and Emukit is successfully used in teaching university-level courses and scientific summer schools, the library still cannot be recommended for absolute beginners.

Emukit puts a strong emphasis on plug-and-play construction of the decision making loop. All components interact via interfaces, and they require a common data format to communicate, which in Emukit is a Numpy array. On one hand Numpy is a defacto standard in scientific Python which means it is reasonable to expect all Emukit users to be able to use Numpy. On the other hand, linear algebra operations in Numpy cannot be GPU-accelerated. This means that while individual components of the outer loop (e.g. a model) can run on GPU, the entire end-to-end process in Emukit is CPU-bound. This issue severely limits Emukit's performance comparing to the libraries that have chosen to rely on a fixed computational backend (BoTorch/PyTorch or Trieste/Tensorflow). This can be potentially mitigated by using specialized libraries that allow Numpy to be run on GPU, such as Numba [46] and CuPy [47].

Conclusions

Emukit is built on a realisation that common methods for decision making under uncertainty – such as Bayesian optimisation, Bayesian quadrature and experimental design – follow the same iterative pattern, and therefore can be seen as instances of a unified high level framework. Emukit provides high level interfaces for these methods that are built on the core set of common abstractions. To enable researchers and practitioners to iterate and experiment quickly Emukit follows plug-and-play design, allowing users to swap out a single part of the decision making loop without affecting other components. Since its initial release in 2019, Emukit has been successfully used in academic research, industry, and teaching.

Emukit has multiple potential growth directions. Mitigation of limitations discussed earlier may improve user experience and overall quality of the library. Integration with other tools in scientific Python ecosystem (e.g. Ray Tune) may increase Emukit's visibility within the community. New functionality, such as multi-objective Bayesian optimisation, would expand library's capabilities and give users new ways to do research with Emukit.

Researchers and enthusiasts from any scientific or industrial domain are welcome to explore the potential of using Emukit for their applications, to contibute new functionality, and to take part in the discussions around the library. The authors are always open to feedback and comments about improvements to the library. The Emukit repository is available on GitHub: https://github.com/EmuKit/emukit.

ACKNOWLEDGMENTS

AP and NL acknowledge the support from the Engineering and Physical Sciences Research Council (EPSRC) and the Alan Turing Institute under grant EP/V030302/1. MM gratefully acknowledges financial support by the European Research Council through ERC StG Action 757275 / PANAMA; the DFG Cluster of Excellence "Machine Learning - New Perspectives for Science", EXC 2064/1, project number 390727645; the German Federal Ministry of Education and Research (BMBF) through the Tübingen AI Center (FKZ: 01IS18039A); and funds from the Ministry of Science, Research and Arts of the State of Baden-Württemberg.

We are thankful to every community member for discussions, comments, bug reports, pull requests, as well as everyone who used the library in their work, study or research. The full list of people who contributed code to Emukit can be found at https://github.com/EmuKit/emukit/graphs/contributors.

REFERENCES

[1] B. Settles, "Active learning literature survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[2] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, ser. Adaptive Computation and Machine Learning. MIT Press, 2006.

[3] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, 1995, pp. 278–282 vol.1, https://doi.org/10.1109/ICDAR.1995.598994.

[4] D. J. C. MacKay, "A practical Bayesian framework for backprop networks," *Neural Computation*, 1991.

[5] A. O'Hagan, M. C. Kennedy, and J. E. Oakley, "Uncertainty analysis and other inference tools for complex computer codes," in *Bayesian Statistics 6*, 1998.

[6] M. C. Kennedy and A. O'Hagan, "Predicting the output from a complex computer code when fast approximations are available," *Biometrika*, vol. 87, no. 1, pp. 1–13, 2000, https://doi.org/10.1093/biomet/87.1.1.

[7] ——, "Bayesian calibration of computer models," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 63, no. 3, pp. 425–464, 2001, https://doi.org/10.1111/1467-9868.00294.

[8] S. Conti, J. P. Gosling, J. E. Oakley, and A. O'Hagan, "Gaussian process emulation of dynamic computer codes," *Biometrika*, vol. 96, no. 3, pp. 663–676, 06 2009, https://doi.org/10.1093/biomet/asp028.

[9] S. Conti and A. O'Hagan, "Bayesian emulation of complex multi-output and dynamic computer models," *Journal of Statistical Planning and Inference*, vol. 140, no. 3, pp. 640–651, 2010, https://doi.org/10.1016/j.jspi.2009.08.006.

[10] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of Bayesian methods for seeking the extremum," *Towards Global Optimization*, vol. 2, no. 117-129, p. 2, 1978.

[11] R. Garnett, *Bayesian Optimization*. Cambridge University Press, 2023, to appear.

[12] A. Baheri and C. Vermillion, "Altitude optimization of airborne wind energy systems: A Bayesian optimization approach," in *2017 American Control Conference (ACC)*. IEEE, 2017, pp. 1365–1370, https://doi.org/10.23919/acc.2017.7963143.

[13] D. E. Graff, E. I. Shakhnovich, and C. W. Coley, "Accelerating high-throughput virtual screening through molecular pool-based active learning," *Chemical science*, vol. 12, no. 22, pp. 7866–7881, 2021, https://doi.org/10.1039/d0sc06805e.

[14] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[15] B. Avent, J. González, T. Diethe, A. Paleyes, and B. Balle, "Automatic discovery of privacy–utility Pareto fronts," *Proceedings on Privacy Enhancing Technologies*, vol. 4, pp. 5–23, 2020, https://doi.org/10.2478/popets-2020-0060.

[16] P. Diaconis, "Bayesian numerical analysis," in *Statistical decision theory and related topics IV*. Springer-Verlag New York, 1988, vol. 1, pp. 163–175.

[17] A. O'Hagan, "Some Bayesian numerical analysis," *Bayesian Statistics*, vol. 4, pp. 345–363, 1992.

[18] P. Hennig, M. A. Osborne, and H. P. Kersting, *Probabilistic Numerics: Computation as Machine Learning*. Cambridge University Press, 2022, https://doi.org/10.1017/9781316681411.

[19] C. E. Rasmussen and Z. Ghahramani, "Bayesian Monte Carlo," in *Advances in Neural Information Processing Systems*, vol. 15, 2002, pp. 505–512.

[20] A. Giovagnoli, "The Bayesian design of adaptive clinical trials," *International Journal of Environmental Research and Public Health*, vol. 18, no. 2, 2021, https://doi.org/10.3390/ijerph18020530.

[21] E. Pauwels, C. Lajaunie, and J.-P. Vert, "A Bayesian active learning strategy for sequential experimental design in systems biology," *BMC Systems Biology*, vol. 8, no. 1, pp. 1–11, 2014, https://doi.org/10.1186/s12918-014-0102-6.

[22] A. E. Gongora, B. Xu, W. Perry, C. Okoye, P. Riley, K. G. Reyes, E. F. Morgan, and K. A. Brown, "A Bayesian experimental autonomous researcher for mechanical design," *Science advances*, vol. 6, no. 15, p. eaaz1708, 2020, https://doi.org/10.1126/sciadv.aaz1708.

[23] B. Peherstorfer, K. Willcox, and M. Gunzburger, "Survey of multifidelity methods in uncertainty propagation, inference, and optimization," *SIAM Review*, vol. 60, no. 3, pp. 550–591, 2018, https://doi.org/10.1137/16M1082469.

[24] The GPy authors, "GPy: A Gaussian process framework in Python," http://github.com/SheffieldML/GPy, 2012.

[25] A. Paleyes, M. Pullin, M. Mahsereci, C. McCollum, N. D. Lawrence, and J. González, "Emulation of physical processes with Emukit," *Second workshop on machine learning and the physical sciences, NeurIPS*, 2019.

[26] E. Siivola, A. Paleyes, J. González, and A. Vehtari, "Good practices for Bayesian optimization of high dimensional structured spaces," *Applied AI Letters*, vol. 2, no. 2, p. e24, 2021, https://doi.org/10.1002/ail2.24.

[27] H. B. Moss, D. S. Leslie, and P. Rayson, "BOSH: Bayesian optimisation by sampling hierarchically," *Workshop on Real World Experimental Design and Active Learning, ICML*, 2020.

[28] M. Naslidnyk, J. Gonzalez, and M. Mahsereci, "Invariant priors for Bayesian quadrature," in *Your Model is Wrong: Robustness and misspecification in probabilistic modeling Workshop, NeurIPS*, 2021.

[29] A. Gessner, J. Gonzalez, and M. Mahsereci, "Active multi-information source Bayesian quadrature," in *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, ser. Proceedings of Machine Learning Research, R. P. Adams and V. Gogate, Eds., vol. 115. PMLR, 2020, pp. 712–721.

[30] S. J. Bell, O. Kampman, J. Dodge, and N. Lawrence, "Modeling the machine learning multiverse," *Advances in Neural Information Processing Systems*, vol. 35, pp. 18 416–18 429, 2022.

[31] A. K. Uhrenholt and B. S. Jensen, "Efficient Bayesian optimization for target vector estimation," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 2661–2670.

[32] D. Liyanage, Y. Ji, D. Everett, M. Heffernan, U. Heinz, S. Mak, and J.-F. Paquet, "Efficient emulation of relativistic heavy ion collisions with transfer learning," *Phys. Rev. C*, vol. 105, p. 034910, Mar 2022, https://doi.org/10.1103/PhysRevC.105.034910. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevC.105.034910

[33] The GPyOpt authors, "GPyOpt: A Bayesian optimization framework in Python," http://github.com/SheffieldML/GPyOpt, 2016.

[34] M. Balandat, B. Karrer, D. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "BoTorch: A framework for efficient Monte-Carlo Bayesian optimization," *Advances in neural information processing systems*, vol. 33, pp. 21 524–21 538, 2020.

[35] V. Picheny, J. Berkeley, H. B. Moss, H. Stojic, U. Granta, S. W. Ober, A. Artemev, K. Ghani, A. Goodall, A. Paleyes *et al.*, "Trieste: Efficiently exploring the depths of black-box functions with Tensorflow," *arXiv preprint arXiv:2302.08436*, 2023.

[36] J. Jiménez and J. Ginebra, "pyGPGO: Bayesian optimization for Python," *Journal of Open Source Software*, vol. 2, no. 19, p. 431, 2017, https://doi.org/10.21105/joss.00431.

[37] G. Louppe, "Bayesian optimisation with scikit-optimize," in *PyData Amsterdam*, 2017.

[38] A. Klein, S. Falkner, N. Mansur, and F. Hutter, "RoBO: A flexible and robust Bayesian optimization framework in Python," in *NIPS 2017 Bayesian Optimization Workshop*, 2017.

[39] OxfordML, "bayesquad," https://github.com/OxfordML/bayesquad, 2013.

[40] J. Wenger, N. Krämer, M. Pförtner, J. Schmidt, N. Bosch, N. Effenberger, J. Zenn, A. Gessner, T. Karvonen, F.-X. Briol, M. Mahsereci, and P. Hennig, "ProbNum: Probabilistic numerics in Python," 2021.

[41] R. D. McMichael, S. M. Blakley, and S. Dushenko, "Optbayesexpt: Sequential Bayesian experiment design for adaptive measurements," *Journal of Research of the National Institute of Standards and Technology*, vol. 126, pp. 1–5, 2021.

[42] Y. Wang, T.-Y. Chen, and D. G. Vlachos, "NEXTorch: a design and Bayesian optimization toolkit for chemical sciences and engineering,"

Journal of Chemical Information and Modeling, vol. 61, no. 11, pp. 5312–5319, 2021, https://doi.org/10.1021/acs.jcim.1c00637.s001.

[43] S. van Rijn and S. Schmitt, "MF2: A collection of multi-fidelity benchmark functions in Python," *Journal of Open Source Software*, vol. 5, no. 52, p. 2049, 2020, https://doi.org/10.21105/joss.02049. [Online]. Available: https://doi.org/10.21105/joss.02049

[44] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019, https://doi.org/10.1145/3292500.3330701.

[45] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[46] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6, https://doi.org/10.1145/2833157.2833162.

[47] R. Nishino and S. H. C. Loomis, "CuPy: A numpy-compatible library for Nvidia GPU calculations," *31st conference on neural information processing systems*, vol. 151, no. 7, 2017.