SoftwareX 11 (2020) 100506

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

BEEP: A Python library for Battery Evaluation and Early Prediction

Patrick Herring [a], Chirranjeevi Balaji Gopal [a], Muratahan Aykol [a,*], Joseph H. Montoya [a], Abraham Anapolsky [a], Peter M. Attia [b], William Gent [b], Jens S. Hummelshøj [a], Linda Hung [a], Ha-Kyung Kwon [a], Patrick Moore [a], Daniel Schweigert [a], Kristen A. Severson [c], Santosh Suram [a], Zi Yang [b], Richard D. Braatz [c], Brian D. Storey [a]

[a] *Toyota Research Institute, Los Altos, CA 94022, USA*
[b] *Department of Materials Science and Engineering, Stanford University, Stanford, CA 94305, USA*
[c] *Department of Chemical Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

A R T I C L E I N F O

Article history:
Received 21 February 2020
Received in revised form 2 April 2020
Accepted 9 April 2020

Keywords:
Battery
Cycling experiments
Python
Data management
Machine-learning

A B S T R A C T

Battery evaluation and early prediction software package (*BEEP*) provides an open-source Python-based framework for the management and processing of high-throughput battery cycling data-streams. BEEPs features include file-system based organization of raw cycling data and metadata received from cell testing equipment, validation protocols that ensure the integrity of such data, parsing and structuring of data into Python-objects ready for analytics, featurization of structured cycling data to serve as input for machine-learning, and end-to-end examples that use processed data for anomaly detection and featurized data to train early-prediction models for cycle life. *BEEP* is developed in response to the software and expertise gap between cell-level battery testing and data-driven battery development.

Code metadata

Current Code version	*1.0*
Permanent link to code/repository used of this code version	https://github.com/ElsevierSoftwareX/SOFTX_2020_49
Legal Code License	*Apache 2.0*
Code Versioning system used	*git*
Software Code Language used	*Python*
Compilation requirements, Operating environments & dependencies	*Python 3 on Linux, OSX or Windows. Dependencies listed in setup.py in code repository.*
If available Link to developer documentation/manual	–
Support email for questions	patrick.herring@tri.global *or* chirranjeevi.gopal@tri.global

1. Motivation and significance

Energy storage in Li-ion batteries revolutionized the portable electronics industry and is now defining the future of vehicle electrification. The growing consumer adoption of electric vehicles (EVs) and the potential for positive environmental benefits have spurred academic and industrial interest in improving the capacity, energy and power density, durability and safety of Li-ion cells, as well as lowering the manufacturing costs [1]. Transitioning to a data-driven research paradigm shows great potential to accelerate battery development –a traditionally slow and tedious process- in areas including the optimization of the chemistry of electrodes, electrolytes, additives [2–5] or formation [6], and designing state-of-health (SoH) [7–10], state-of-charge (SoC) [10–12], early prediction models [9,10,13] or advanced battery management systems (BMSs) [14–16]. An increase in the volume of standardized cycling data can open the door to improvement of existing approaches to data-driven prognostics, [9,13,15,17] or to design of more complex algorithms capable of delivering accurate health predictions.

For the broader adoption of data-driven battery development, reusable high-throughput battery testing data [13,18] and software tools for processing and analysis of such data are essential. While the hardware for automated battery cycling is accessible, the field still lacks open software for both acquisition and management of cycling data and preparing the data for analytics.

* Correspondence to: 4440 El Camino Real, Los Altos, CA 94022, USA.
E-mail address: murat.aykol@tri.global (M. Aykol).

https://doi.org/10.1016/j.softx.2020.100506

Community-driven software development can be effective in filling this gap [19] and can yield reliable and reusable tools (as experienced in computational materials science [20,21]). Such software libraries can form the basis for advanced development capabilities for the expert and lower the barrier for the novice to set foot in data-centric battery research.

The repetitive nature of battery experiments defines the requirements for such a tool to be useful to battery researchers. Experiments consist of repeated application of "cycling protocols" (which prescribe how the battery should be charged and discharged) to a user-supplied battery cell by the hardware. Such cycling experiments can take from a few hours to several months to complete. During these experiments, many types of information, such as time, capacity, voltage, cycle number and temperature are recorded with high sampling frequency, and the size of raw data can grow rapidly. Besides, the naming conventions for different projects, data, metadata and protocol files can vary among vendors or members of a research group. Hence, a scalable data management and processing system is required. In addition, the data structures can be complicated due to the cyclic nature of the experiments. For example, raw data may need to be grouped over one axis and interpolated over another. Data formats and storage technologies (e.g. databases, file systems) also vary among different cycler hardware. Researchers could benefit from standardized data formats, alongside programmatic interfaces to such databases as needed. Data streams should be validated against human errors, equipment errors or failures, and environmental circumstances to ensure their integrity. Organized, processed, and validated data are the key ingredient for a data-driven research pipeline, and can be used in unsupervised modeling (e.g. for anomaly detection) or followed by a "featurization" step that computes engineered features from the data [13,22]. Featurized cycling data can be used in training predictive models (e.g. for failure prediction).

To the best of our knowledge, there is no open software that satisfies the requirements or features summarized above, which are expected to be useful for enabling wider adoption of data-driven approaches in battery research. The battery experimentation and early prediction Python library, BEEP, aims to fill this gap. Since it is built on common Python libraries such as NumPy, SciPy, scikit-learn and pandas, and adopts common data interchange formats like JSON, we expect BEEP to make this transition to data-driven research easier for individual researchers and provide useful building blocks for battery research platforms developed by research groups [23].

2. Software description

BEEP consists of six main modules: `collate`, `validate`, `structure`, `featurize`, `generate_protocol` and `run_model`. While the modules can be used independently, this order of execution is typical for the data management and processing steps delivered by the BEEP framework, and therefore the output of the main methods in each module is the input for the next. The default functionality of each module can also be directly accessed from the command-line, where input arguments are provided as JSON-strings. Almost all BEEP classes are serializable and can be stored as such objects. Here we explain the main functionalities delivered with each module in BEEP.

2.1. Collate

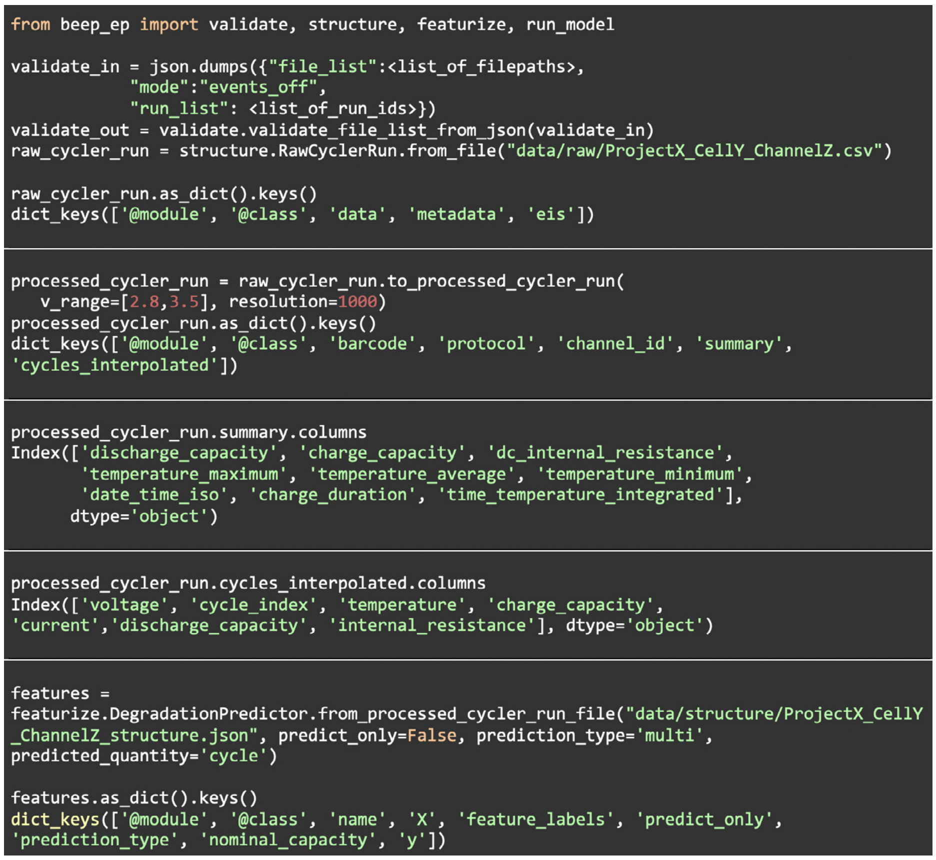
The `collate` module is used for standardization of raw cycler files and metadata as well as organization of the standardized files. BEEP follows a name-based convention for file storage and all paths are defined in reference to the `BEEP_EP_ROOT` environment variable. If `collate` is called from the command line, the module locates the raw data files, parses the metadata, and collates files according to a combination of protocol, channel number, and date, organizing them in '`/data-share/collated_cycler_files`'. This functionality is handled mainly by the function `process_files_json` which can also be called directly. The output is a JSON string that contains ids, paths and names for raw cycler files, paths for the collated cycler files, cycling protocols corresponding to each file, channel number and the date the original file was generated.

2.2. Validate

As in any experimental process, erroneous or corrupted data can be produced as a consequence of instrument failures (e.g. power outage), changes in environmental conditions (e.g. temperature), software glitches or human errors (e.g. misconfigured protocols) in any stage of battery cycling experiments. If unnoticed, such data may contaminate the analytical process or misguide the research. To address this issue, BEEP provides a `validate` module, where the `ValidatorBeep` class validates collated cycling data against researcher-defined schemas prescribed in yaml files (examples can be found in `VALIDATION_SCHEMA_DIR`) or as dictionary-based rule definitions of the external Python library `cerberus` [24]. The validation schemas can include data-types, min/max values, ranges, non-allowed values or complex rules via `cerberus`, which adopts a convenient, dictionary based schema definition. A fast, lightweight version is provided as `SimpleValidatorBeep` which does dataframe-based validation (restricted to type, min/max and non-allowed) and supports the `cerberus` syntax for interchangeable use. Validation stores the list of files being validated and the results in JSON format, at `DEFAULT_VALIDATION_RECORDS`.

2.3. Structure

Battery cycling tests accumulate information in a tabular form, containing thousands to millions of rows, and produce large data files that need to be structured for analytics. The `structure` module contains two classes that serve this purpose: `RawCyclerRun` and `ProcessedCyclerRun`. The first class supports parsing and indexing of raw data into appropriate integer (e.g. step, cycle index) and float (e.g. time, current, voltage, charge capacity, temperature) columns in a dataframe, and provides methods to identify diagnostic cycles, and deliver summary statistics and metadata. This class can interpolate target variables over other variables and return interpolated data-containers of the same structure (e.g. interpolating variables on a consistent voltage scale), which is useful for machine-learning models. `ProcessedCyclerRun` provides project-specific structuring of raw data from `RawCyclerRun`, for which example schemas are provided in the `conversion_schemas` folder for various types of hardware. Input data needed for structuring exist in datafiles of almost any cycling hardware, often recorded with different naming conventions. This library of conversion schemas can be expanded to other formats and provide a centralized resource for the community to be used with other battery cyclers and instruments. The `ProcessedCyclerRun` class produces a rich, serializable object, which flexibly allows addition of fields and data as needed.



```
from beep_ep import validate, structure, featurize, run_model

validate_in = json.dumps({"file_list":<list_of_filepaths>,
            "mode":"events_off",
            "run_list": <list_of_run_ids>})
validate_out = validate.validate_file_list_from_json(validate_in)
raw_cycler_run = structure.RawCyclerRun.from_file("data/raw/ProjectX_CellY_ChannelZ.csv")

raw_cycler_run.as_dict().keys()
dict_keys(['@module', '@class', 'data', 'metadata', 'eis'])
```

```
processed_cycler_run = raw_cycler_run.to_processed_cycler_run(
   v_range=[2.8,3.5], resolution=1000)
processed_cycler_run.as_dict().keys()
dict_keys(['@module', '@class', 'barcode', 'protocol', 'channel_id', 'summary',
'cycles_interpolated'])
```

```
processed_cycler_run.summary.columns
Index(['discharge_capacity', 'charge_capacity', 'dc_internal_resistance',
       'temperature_maximum', 'temperature_average', 'temperature_minimum',
       'date_time_iso', 'charge_duration', 'time_temperature_integrated'],
      dtype='object')
```

```
processed_cycler_run.cycles_interpolated.columns
Index(['voltage', 'cycle_index', 'temperature', 'charge_capacity',
'current','discharge_capacity', 'internal_resistance'], dtype='object')
```

```
features =
featurize.DegradationPredictor.from_processed_cycler_run_file("data/structure/ProjectX_CellY
_ChannelZ_structure.json", predict_only=False, prediction_type='multi',
predicted_quantity='cycle')

features.as_dict().keys()
dict_keys(['@module', '@class', 'name', 'X', 'feature_labels', 'predict_only',
'prediction_type', 'nominal_capacity', 'y'])
```

Fig. 1. Code snippets demonstrating the raw data file handling, processing and featurization.

2.4. Featurize

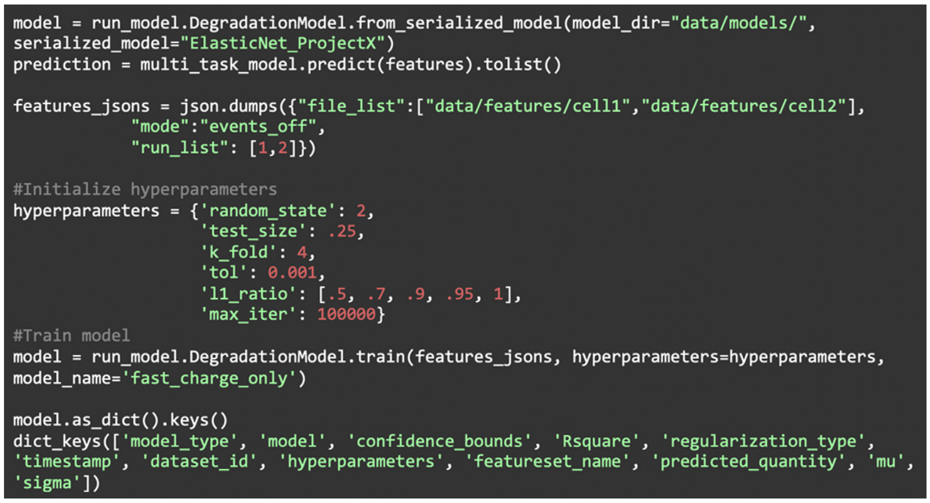
Data input for machine learning algorithms generally needs to be uniformly formatted. For many cases, this formatting can be done through the construction of *features*, which are quantities computed from raw or interpolated data, and are based on known physical phenomena in the system. We handle this process in the `Featurize` module. In the `DegradationPredictor` class, features are computed on a per-cell basis from `ProcessedCyclerRun` and are used to predict the performance of the battery at a certain number of cycles in the future. Many of these quantities could be computed per cycle or at specific times in order to fit a model that predicts the performance of the battery at a point in the future (relative or absolute). Currently default features include time integrated temperature at cycle 100, capacity decrease over the first 100 cycles, minimum temperature in the first 100 cycles, and others [13]. Additional features can be easily added, but care must be taken to also update the downstream models. Community improvement of this feature set is a desirable development direction.

2.5. Model

The `model` module comprises methods to aggregate featurized battery cycling data, and train and store machine-learning models, currently for early prediction of cycle life. As input, it takes feature files encoding `DegradationPredictor` objects. The core of the module is `DegradationModel` and its associated attributes/methods for model initialization, hyperparameter tuning, training and cross-validation, and predictions. Users have the option to load existing serialized models, or train new models on the set of `DegradationPredictor` objects using the `train` method. Single or multi-point fitting can be performed, and `train` automatically makes that determination from the dataset. Model attributes, coefficients (for linear models), performance metrics, dataset-id and other metadata required to reproduce the training are serialized as a JSON file. The predictions are reported as cycle-life, or the number of cycles to reach a certain discharge capacity relative to the nominal value, along with a 95% confidence interval. Current implementation includes regularized multi-linear regression, but as the model objects build on scikit-learn (and hence use its estimator API), they are easily extensible to other machine learning models, like ensemble methods or neural networks in scikit-learn or similar libraries. Integration with other machine-learning libraries can be achieved by adopting their APIs in new model classes derived from existing ones above.

2.6. Protocol

Most battery cycling systems have a "protocol" file that is used to run the cycling experiment. This file contains parameters that control the cell, limit conditions for each of the steps in the cycling experiment, and other variables. Different hardware vendors refer to these files differently, e.g. as schedule, procedure, or sequence files and adopt different formats in terms of language and layout. We use the term protocol to refer to these files and provide an abstraction for their components to unify such different formats. In a manner similar to the data files, we structure different protocol formats into JSON objects that can be accessed and modified. This allows programmatic generation of protocols in the `generate_protocol` module. We provide functions that convert protocol objects to the file format in use. Care should be taken when using this functionality since there are numerous



```
model = run_model.DegradationModel.from_serialized_model(model_dir="data/models/",
serialized_model="ElasticNet_ProjectX")
prediction = multi_task_model.predict(features).tolist()

features_jsons = json.dumps({"file_list":["data/features/cell1","data/features/cell2"],
            "mode":"events_off",
            "run_list": [1,2]})

#Initialize hyperparameters
hyperparameters = {'random_state': 2,
                   'test_size': .25,
                   'k_fold': 4,
                   'tol': 0.001,
                   'l1_ratio': [.5, .7, .9, .95, 1],
                   'max_iter': 100000}
#Train model
model = run_model.DegradationModel.train(features_jsons, hyperparameters=hyperparameters,
model_name='fast_charge_only')

model.as_dict().keys()
dict_keys(['model_type', 'model', 'confidence_bounds', 'Rsquare', 'regularization_type',
'timestamp', 'dataset_id', 'hyperparameters', 'featureset_name', 'predicted_quantity', 'mu',
'sigma'])
```

Fig. 2. Code snippets demonstrating model prediction using existing (stored) model and training of a new model using featurized data.

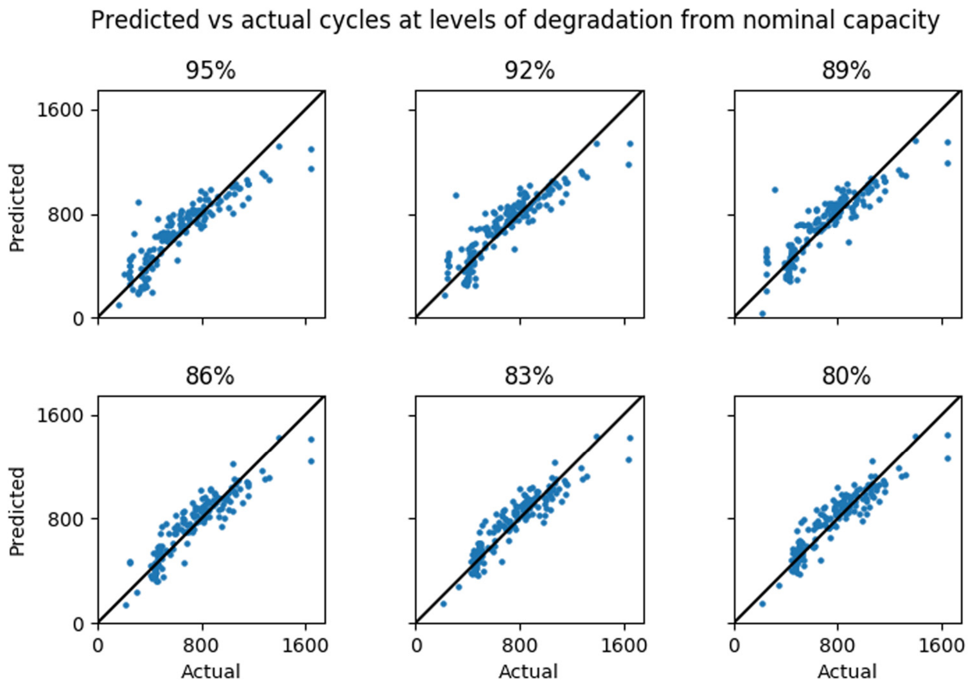


Fig. 3. Predicted vs. actual cycle life at different capacity fade thresholds. The capacity fade threshold is shown as a percentage of initial nominal capacity in each panel.

safety values included in the file; setting these values incorrectly can result in damage to the cycling system or the cell under test, either of which might be catastrophic. However, the benefits of this capability are far reaching, such as the elimination of manual test creation, which is time-consuming and error-prone, and the ability to create and run protocols in cyclers with minimal human input. These features enable automated control and selection of experiments by active learning systems, as demonstrated in Ref. [25]. Future developments include the ability to convert from one system's protocol file to another's (within the appropriate hardware constraints) and supporting more systems.

2.7. Other features

BEEP scripts can be run locally on a machine with appropriate access to data and adequate compute/memory. But dealing with the computational workload of data processing, model fitting and display rendering can become a roadblock in large battery cycling experiments. Hence, components of BEEP are modularized for easy deployment to cloud-based services that can scale up or down. Additional cloud infrastructure can provide messaging and coordination of various scripts to deal with large data loads and processing-heavy tasks. For such application environments, we designed most of the scripts to be containerized and run via command-line arguments, and messaging between components to be achieved with event streaming.

Currently, BEEP assumes that data are arriving in a flat-file format, with one file per test. There are, however, database-centered cycler systems that do not store or export data in this form, which improves performance but makes data less accessible. Scripts that enable integration with such systems and output of flat files ingestible by BEEP are available at https://github.com/TRI-AMDD/beep-integration-scripts. There are two caveats to such scripts. First, there are assumptions about the way that the tests are run, such as the user will not duplicate a test name on a channel, each test runs to completion on a single channel, etc. These assumptions stem from common practices, but the scripts have not been tested against a large number of conditions. Second, while for most cases the data files are less than a few hundred megabytes, sufficient memory must be available to fit large tests into memory. Further development might reduce the memory requirement and provide more robust extraction of the data, e.g. with improved SQL queries.

3. Illustrative example

BEEP is intended to streamline the process of transforming raw data from a battery cycler into actionable insights through data management, transformation and modeling. As a simplified end-to-end pipeline, here we will parse the raw dataset published by Severson et al. [13], and featurize it to train a multi-task linear model for early-prediction of cycle life. Relevant code snippets for reproducing this exercise are provided in Figs. 1 and 2. First, the paths to desired raw cycling data are compiled into a JSON object, and validated, the output of which is a JSON object containing the validity information. Next, the validated raw data are structured in two steps. First, a `RawCyclerRun` object is initialized to store time-series of measured quantities as a dataframe in `data` and `metadata`, and electrochemical impedance spectra in `eis`, if measured. The raw data are then processed to yield a `ProcessedCyclerRun` object, with optional arguments specifying the range and resolution of voltage interpolation. The resulting object contains dataframes that store summary statistics, and interpolated discharge curves. The object also contains meta-data, such as the `barcode` of the cell, `protocol` and `channel_id`. Each row in `summary` dataframe stores point-measures of various metrics for a cycle. The resulting dataframe is useful for developing features that rely on aggregated properties over a cycle. The `cycles_interpolated` dataframe contains similar quantities as `summary`, but instead of aggregation over cycles, the values are interpolated on an evenly spaced voltage grid within each cycle. This dataframe is useful for differential analysis at specific voltages, or for featurization based on material-specific properties not captured in `summary`. The processed cycle run can be used to prepare features via `DegradationPredictor`, with optional arguments specifying the quantity to be predicted (e.g. `cycle` as a function of `capacity`) and type (`single` vs. `multi` task). The feature object contains the name of the feature-set, actual values of the features in a dataframe `X`, feature names in `feature_labels`, `nominal_capacity` of the cell in the first few cycles and the outcomes `y` for training data.

In Fig. 2, we show how a previously trained and stored model (`DegradationModel`) can be loaded to make predictions on incoming featurized-data or how a new model can be trained from scratch on a set of `DegradationPredictor` objects, using the `train` method and a dictionary of hyperparameters for optimization. The `train` method assembles all the predictors into a dataframe, performs train-test-split, hyperparameter optimization and cross-validation as specified by `hyperparameters`. Following that, predictions can be generated and/or the `DegradationModel` object can be serialized and stored with all requisite details to reproduce the training process at a later time. In the current implementation `model` key corresponds to a linear model and `confidence_bounds` is the 95% confidence interval (for each task) calculated on the test data. A sample plot of actual vs. predicted capacity fade using a multi-task model trained on 4 different batches of cells is shown in Fig. 3. Specifically, the cycles (time) taken to reach 95%, 92%, 89%, 86%, 83% and 80% of nominal capacity using the first 100 cycles are predicted. Predictions of the model are further compared with the experimental capacity fade curves in Fig. 4.

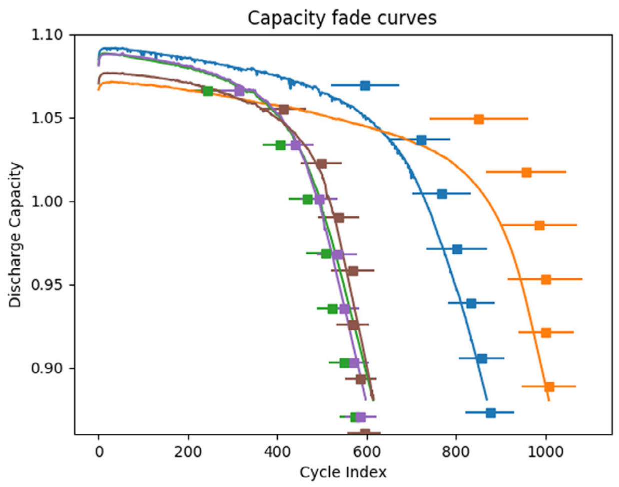


Fig. 4. Capacity fade curves (discharge capacity vs. cycle index). Experimental measurements reported in Severson et al. [13] are shown as solid lines, and predictions made using the model illustrated in the text are shown as points, along with the associated standard deviations as horizontal bars.

4. Impact

As the use of Li-ion batteries grows, especially in the EV market, the importance of understanding their potential and limitations will increase rapidly. We view BEEP as a platform that can accelerate battery research by removing the burden of data organization from the researcher and automating as many processes as possible. BEEP enables researchers to efficiently deal with larger sample sizes, improving the repeatability and reliability of their results. Automated organization, cleaning and structuring of data make sharing and collaboration seamless, and machine-learning approaches more accessible for researchers. Our hope is that as the battery community builds larger public datasets, better and more complex models can be trained and these models can guide battery development and reduce the number of physical battery experiments, accelerating the pace of battery development. Such models can potentially provide more accurate predictions, e.g. for diagnostics, health management or future performance. Our goal with BEEP is to create a community composed of both academic and industrial entities, dedicated to building methods for data-driven battery development.

5. Conclusion

We present a set of methods for automated ingestion and analysis of battery cycling data. These methods are embodied in Python scripts that leverage open source formats and packages to allow a larger community of battery researchers to use them.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: P.K., M.A., A.A., C.G., P.A. R.B., and K.S. have U.S. patent applications in the area of battery data management and prognostics. The remaining authors have no competing interests to declare.

Acknowledgments

Authors acknowledge Prof. William Chueh and Prof. Martin Bazant for the discussions throughout the development of BEEP.

References

[1] Thackeray MM, Wolverton C, Isaacs ED. Energy Environ Sci 2012;5:7854.
[2] Kauwe S, Rhone T, Sparks T. Crystals 2019;9:54.
[3] Halls MD, Tasaki K. J Power Sources 2010;195:1472–8.
[4] Qu X, Jain A, Rajput NN, Cheng L, Zhang Y, Ong SP, et al. Comput Mater Sci 2015;103:56–67.
[5] Hautier G, Jain A, Chen H, Moore C, Ong SP, Ceder G. J Mater Chem 2011;21:17147.
[6] Ermon S, Chueh WC, Grover A, Markov TM, Perkins N, Attia PM. Autonomous screening and optimization of battery formation and cycling procedures. USPTO No. US 2019/0115778 A1, 2019.
[7] Shen S, Sadoughi M, Chen X, Hong M, Hu C. J Energy Storage 2019;25. 100817.
[8] Richardson RR, Osborne MA, Howey DA. J Energy Storage 2019;23:320–8.
[9] Li Y, Liu K, Foley AM, Zülke A, Berecibar M, Nanini-Maury E, et al. Renew Sustain Energy Rev 2019;113.
[10] Plett GL. J Power Sources 2004;134:277–92.
[11] Ng KS, Moo CS, Chen YP, Hsieh YC. Appl Energy 2009;86:1506–11.
[12] Xing Y, He W, Pecht M, Tsui KL. Appl Energy 2014;113:106–15.
[13] Severson KA, Attia PM, Jin N, Perkins N, Jiang B, Yang Z, et al. Nat Energy 2019;4:383–91.
[14] Lu L, Han X, Li J, Hua J, Ouyang M. J Power Sources 2013;226:272–88.
[15] Pecht M, Jaai R. Microelectron Reliab 2010;50:317–23.
[16] Rezvanizaniani SM, Liu Z, Chen Y, Lee J. J Power Sources 2014;256:110–24.
[17] Nuhic A, Terzimehic T, Soczka-Guth T, Buchholz M, Dietmayer K. J Power Sources 2013;239:680–8.
[18] Wilkinson MD, Dumontier M, Aalbersberg IjJ, Appleton G, Axton M, Baak A, et al. Sci Data 2016;3:1–9.
[19] Ward L, Aykol M, Blaiszik B, Foster I, Meredig B, Saal J, et al. MRS Bull 2018;43:683–9.
[20] Ong SP, Richards WD, Jain A, Hautier G, Kocher M, Cholia S, et al. Comput Mater Sci 2013;68:314–9.
[21] Kirklin S, Saal JE, Meredig B, Thompson A, Doak JW, Aykol M, et al. Npj Comput Mater 2015;1:15010.
[22] Ward L, Dunn A, Faghaninia A, Zimmermann NER, Bajaj S, Wang Q, et al. Comput Mater Sci 2018;152:60–9.
[23] Aykol M, Hummelshøj JS, Anapolsky A, et al. Matter 2019;1:1433–8.
[24] Cerberus Python package. 2020, http://python-cerberus.org (accessed April 1, 2020).
[25] Attia P, Grover A, Severson K, et al. Nature 2020;578:397–402.