On the Creation of a Chess-AI-Inspired Problem-Specific Optimizer for the Pseudo Two-Dimensional Battery Model Using Neural Networks

Neal Dawson-Elli,[1] Suryanarayana Kolluri,[1,*] Kishalay Mitra,[2] and Venkat R. Subramanian[1,3,,z]**

[1]*Department of Chemical Engineering, University of Washington, Seattle, Washington 98195, USA*
[2]*Indian Institute of Technology Hyderabad, Kandi, Sangareddy, Medak, 502 285 Telangana, India*
[3]*Pacific Northwest National Laboratory, Richland, Washington 99352, USA*

In this work, an artificial intelligence based optimization analysis is done using the porous electrode pseudo two-dimensional (P2D) lithium-ion battery model. Due to the nonlinearity and large parameter space of the physics-based model, parameter calibration is often an expensive and difficult task. Several classes of optimizers are tested under ideal conditions. Using artificial neural networks, a hybrid optimization scheme inspired by the neural network-based chess engine DeepChess is proposed that can significantly improve the converged optimization result, outperforming a genetic algorithm and polishing optimizer pair by 10-fold and outperforming a random initial guess by 30-fold. This initial guess creation technique demonstrates significant improvements on accurate identification of model parameters compared to conventional methods. Accurate parameter identification is of paramount importance when using sophisticated models in control applications.

Manuscript submitted January 17, 2019; revised manuscript received March 1, 2019. Published March 15, 2019.

Lithium ion batteries are complex electrochemical devices whose performance is dictated by design, thermodynamic, kinetic, and transport properties. These relationships result in nonlinear and complicated behavior, which is strongly dependent upon the conditions during operation. Despite these complexities, lithium ion batteries are nearly ubiquitous, appearing in cell phones, laptops, electric cars, and grid-scale operations.

The battery research community is continually seeking to improve models which can predict various states of the battery. Due to this drive, a multitude of physics-based models which aim to describe the internal processes of lithium ion batteries can be found, ranging in their complexity and accuracy from computationally expensive molecular dynamics simulations down to linear equivalent circuit and empirical models. Continuum scale models such as the single particle model[1] and pseudo two-dimensional model (P2D)[2–5] exist between these extremes and trade off some physical fidelity for decreased execution time. These continuum models are generally partial differential equations, which must be discretized in space and time in order to be solved. In an earlier work, a model reformulation based on orthogonal collocation[2] was used to greatly decrease solve time while retaining physical validity, even at high currents.

Sophisticated physics-based battery models are capable of describing transient battery behavior during dynamic loads. Pathak et al.[6] have shown that by optimizing charge profiles based on internal model states, real-world battery performance can be improved, doubling the effective cycle life under identical charge time constraints in high current applications. Other work[7] has shown similar results, lending more credence to the idea that modeled internal states used as control objectives can improve battery performance. However, in order for these models to be accurate, they must be calibrated to the individual battery that is being controlled. This estimation exercise is a challenging task, as the nonlinear and stiff nature of the model coupled with dynamic parameter sensitivity can wreak havoc on black-box optimizers. Going into different approaches for estimating parameters is beyond the scope of this paper.

Data science, often hailed as the fourth paradigm of science,[8] is a large field, which covers data storage, data analysis, and data visualization. Machine learning, a subfield of data science, is an extremely flexible and powerful tool for making predictions given data, with no explicit user-parameterized model connecting the two. Artificial neural networks, and the most popular form known as deep neural networks (DNNs), are extremely powerful learning machines which can approximate extremely complex functions. Previous work has looked at examining the replacement of physics-based models with decision trees, but this has proven to be moderately effective, at best.[9–12] In this work, neural networks are used not to replace the physics-based model, but to assist the optimizer. The flexibility in problem formulation of neural networks affords the ability to map any set of inputs to another set of outputs, with statistical relationships driving the resulting accuracy. In this instance, the aim is to use the current parameter values coupled with the value difference between two simulated discharge curves to refine the initial parameter guess and improve the converged optimizer result. The outputs of the neural network will be the correction factor which, when applied to the current parameter values, will approximate the necessary physics-based model inputs to represent the unknown curve.

One interesting aspect of data science is the idea that it is difficult to create a tool which is only valuable in one context. For example, convolutional neural networks were originally created for space-invariant analysis of 2D image patterns,[13] and have been very successful in image classification and video recognition,[14] recommender systems,[15] medical image analysis,[16] and natural language processing.[17] To this end, the comparative framework inspired by DeepChess[18] has found effective application in the preprocessing and refinement of initial guesses for optimization problems using physics-based battery models, where the end goal of the optimization is to correctly identify the model parameters that were used to generate the target curves. Specifically, the problem formulation is inspired by DeepChess – the ideas of shuffling the data each training epoch and of creating a comparative framework were instrumental, and the inspiration is the namesake of this work. The problem of accurate parameter identification and model calibration is paramount if these sophisticated models are to find applications in control and design. The process of sampling the model, creating the neural networks, and analyzing the results are outlined for a toy 2-dimensional problem using highly-sensitive electrode thicknesses and a 9-dimensional problem, where the parameters vary significantly in sensitivity, scale, and boundary range.

In this article, the sections are broken up as follows: the problem formulation and an overview of neural networks are discussed first, followed by the sensitivity analysis and a two-dimensional demonstration of the neural network formulation. Then, the problem is discussed in 9 dimensions, where the neural network recommendation is restricted

*Electrochemical Society Student Member.
**Electrochemical Society Fellow.
[z]E-mail: vsubram@uw.edu

to a single function call, meaning that the neural network performs a single refinement of an initial guess, and different optimization algorithms are explored. In the next section, the same 9 dimensions are used, but the neural network is randomly sampled up to 100 times, and the best guess from the resulting refinements is used in the optimization problem. This is shown to dramatically improve the converged result. The resulting neural network refinement system is only applicable to this model, with these varied parameters, over these bounds and at the sampled current rates, which effectively makes the neural network a problem-specific optimizer.

Ideas of Data Science and Problem Formulation

Traditionally, data science approaches to optimization or fitting problems would include creating a forwards or inverse model, as has already been explored using decision trees.[9] In the forwards problem formulation, a machine learning algorithm would map the target function inputs to the target function outputs, and then traditional optimization frameworks would use this faster, approximate model rather than the original expensive high fidelity model, with the optimization scheme remaining unchanged. However, this is not the most efficient way to use the data that has been generated, and comes with additional difficulties, including physical consistency.[19] The concerns with physical consistency stem from machine learning algorithms' native inability to enforce relative output values, which can violate output conditions like monotonicity or self-consistency.

An additional problem formulation style, known as inverse problem formulation, seeks to use machine learning in order to map the outputs of a function to its inputs, seeking to create an O(1) optimization algorithm tailored to a specific problem. While this ambitious formulation will always provide some guess as to model parameters, they are subject to sensitivity and sampling considerations and a large amount of data is needed for these inverse algorithms to be successful in nonlinear systems.[9]

In this work, the flexibility of neural network problem formulation is leveraged in order to create a comparative framework. A comparative framework could be considered a way of artificially increasing the size of the data set, one of several tricks which are common in the machine learning space known as data augmentation.[20] Another popular technique, especially in 2-dimensional image classification, is subsampling of an image with random placement. This forces the neural network to generalize the prediction with respect to the location of learned features and, more importantly, allows the neural network to leverage more information from the same data set by creating more unique inputs to the model.

In order to make the model nonlinear, activations functions are applied element-wise during neural network calculations. In this work, an exponential linear unit (eLU)[21] was used, the input-output relationship of which is described in Figure 1. This activation function was chosen to keep the positive attributes of rectified linear units (ReLU)[22] while preventing ReLU die-off associated with zero gradients at negative inputs. Another consideration when selecting eLU as the activation function was to avoid the "vanishing gradient problem".[23] During training, error is propagated as gradients from the last layers of the neural network back to the top layers. When using activation functions which can saturate, such as sigmoids or hyperbolic tangents, the gradients passed through these activation functions can be very small if the activation function is saturated. This effect is compounded as the depth of the neural network increases. By selecting an activation function which can only saturate in one direction, and by restricting the depth of the neural network to only three layers, vanishing gradients can be avoided.

Additionally, each of the neural networks used in this work are of identical size, consisting of hidden layers of 95, 55, and 45 nodes, and each neural network was trained for a maximum of 500 epochs. Mean squared error was used as the loss function for each neural network, and Adam[24] was the training optimizer of choice.

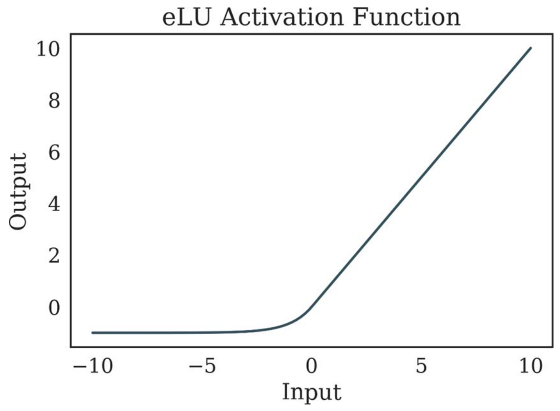


Figure 1. Exponential Linear Unit activation function input-output mapping. The activation function adds nonlinearity to the neural network which allows the outputs to capture more complex phenomena.

2D Application and Sensitivity Analysis

The version of the P2D model used in this work has 24 parameters which can be modified in order to fit experimental data; a set composed of transport and kinetic constants. In order to reduce this large dimensionality to something more reasonable, a simple one-at-a-time sensitivity analysis was performed. A range for each parameter was established based on a combination of literature values[9] and model solution success, and an initial parameter set was chosen. For each of the parameters, the values were permuted from the initial value to the upper and lower bounds. The resulting change in root mean squared error (RMSE) was used as the metric for sensitivity. The results are summarized along with the bounds in Table I and Figure 2. These sensitivities were used to inform the parameter down-sampling to 9 dimensions. The selected parameters are bolded in Table I. The diffusivities were selected for their relatively low sensitivity and wide bounds, while the maximum concentrations in the positive and negative electrodes, porosity in positive and negative electrodes, and thicknesses of positive

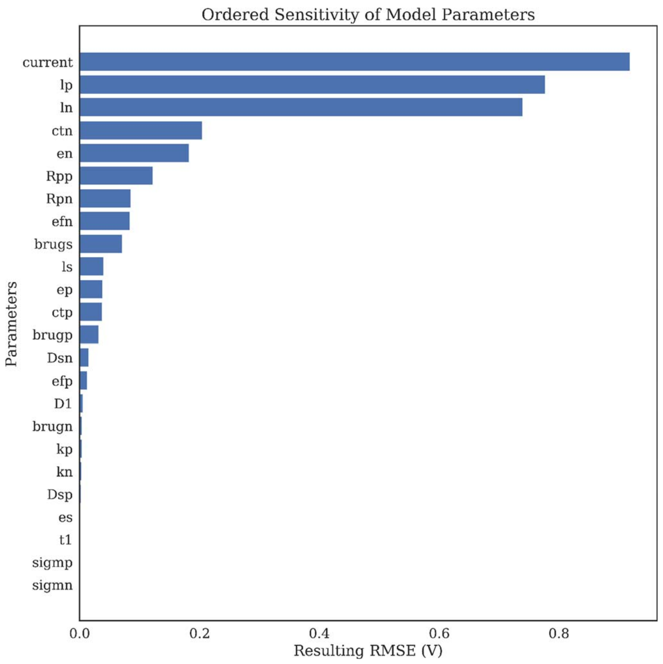


Figure 2. Ordered sensitivity of model parameters, calculated by measuring the root mean squared error change in output voltage associated with a unit step of 10% of the bounded range in either direction from the initial value. It is important to note that this is a function of the current parameter values.

Table I. Sensitivity and bounds of available P2D Model parameters. Bolded values are selected for use in the 9-dimensional analysis which makes up the bulk of this work.

Parameter	Description	Lower Bound	Upper Bound	Sensitivity	Units
D1	**Electrolyte Diffusivity**	**7.50E-11**	**7.50E-09**	**0.0053579**	$\mathbf{\frac{m^2}{s}}$
Dsn	**Solid-phase diffusivity (n)**	**3.90E-15**	**3.90E-13**	**0.0143956**	$\mathbf{\frac{m^2}{s}}$
Dsp	**Solid-phase diffusivity (p)**	**1.00E-15**	**1.00E-13**	**0.001759**	$\mathbf{\frac{m^2}{s}}$
Rpn	Particle radius (p)	2.00E-07	2.00E-05	0.0841821	m
Rpp	Particle radius (n)	2.00E-07	2.00E-05	0.1211896	m
brugp	Bruggeman coef (p)	3.6	4.4	0.0312093	
brugs	Bruggeman coef (s)	3.6	4.4	0.0702692	
brugn	Bruggeman coef (n)	3.6	4.4	0.0032529	
ctn	**Maximum solid phase concentration (n)**	**27499.5**	**33610.5**	**0.2041194**	$\mathbf{\frac{mol}{m^3}}$
ctp	**Maximum solid phase concentration (p)**	**46398.6**	**56709.4**	**0.0373641**	$\mathbf{\frac{mol}{m^3}}$
efn	Filler fraction (n)	0.02934	0.03586	0.0827452	
efp	Filler fraction (p)	0.0225	0.0275	0.012084	
en	Porosity (n)	0.4365	0.5335	0.1816505	
ep	**Porosity (p)**	**0.3465**	**0.4235**	**0.0379277**	
es	**Porosity (s)**	**0.6516**	**0.7964**	**0.0010536**	
iapp	Current density	13.5	16.5	0.9174174	$\frac{A}{m^2}$
kn	Reaction rate constant (n)	5.03E-12	5.03E-10	0.0028494	$\frac{\frac{mol}{(s\ m^2)}}{(\frac{mol}{m^3})^{1+\alpha a,i}}$
kp	Reaction rate constant (p)	2.33E-12	2.33E-10	0.0032171	$\frac{\frac{mol}{(s\ m^2)}}{(\frac{mol}{m^3})^{1+\alpha a,i}}$
lp	**Region thickness (p)**	**8.80E-06**	**0.00088**	**0.7764841**	$\mathbf{m}$
ln	**Region thickness (n)**	**8.00E-06**	**0.0008**	**0.7386793**	$\mathbf{m}$
ls	Region thickness (s)	2.50E-06	0.00025	0.0398143	m
σ_n	Solid-phase conductivity (n)	90	110	7.45E-06	$\frac{S}{m}$
σ_p	Solid-phase conductivity (p)	9	11	1.74E-05	$\frac{S}{m}$
t1	Transference number	0.3267	0.3993	8.88E-05	

and negative electrodes were selected for a combination of high sensitivity and symmetry. In general, it would be advisable to select only the most sensitive parameters when fitting, regardless of symmetry. However, since these sensitivities are only locally accurate, symmetry was prioritized above absolute sensitivity for interpretability. The diffusivities were selected to examine how the DNN performs when guessing parameters with low sensitivity.

The selected parameters, with the exception of the diffusivities, were varied uniformly across their bounds in accordance with a quasi-random technique known as Sobol sampling.[25] The diffusivities had a very large parameter range, roughly two orders of magnitude, so log scaling was used before a uniform distribution was applied, which ensured that the sampling was not dominated by the upper values from the range. The main benefit of the Sobol sampling technique is that it more uniformly covers a high dimensional space than naïve random sampling without the massive increase in sampling size required to perform a rigorous factorial sweep. The downside is that it is not as statistically rigorous as other techniques like fractional factorial or Saltelli sampling, which have the added advantages of allowing for sensitivity analyses.[26] However, the number of repeated levels for each parameter is much higher with these methods of sampling, which results in worse performance when used as the sampling technique for training a neural network.

Two of the most sensitive parameters, the thicknesses of the negative and positive electrodes, were selected to demonstrate the non-convexity of the optimization problem while performing parameter estimation for the P2D model. A relatively small value range was selected, and 50 discrete levels for each parameter were chosen, resulting in 2500 discharge curves, which were generated using each of the values. Then, a random, near-central value was selected as the 'true value,' corresponding to 0 RMSE, and an error contour plot was created as shown in Figure 3. The blue dot represents the true values of parameters and an RMSE of 0. As these discharge curves are simulated, their final times can vary. In order to make the two curves comparable, the target curve is padded with values equal to the final voltage of 2.5 V, and any curve which terminates early is also padded with values equal to 2.5 V. Another option would be to simply interpolate at the target data points and throw away any data past either terminal time, but this

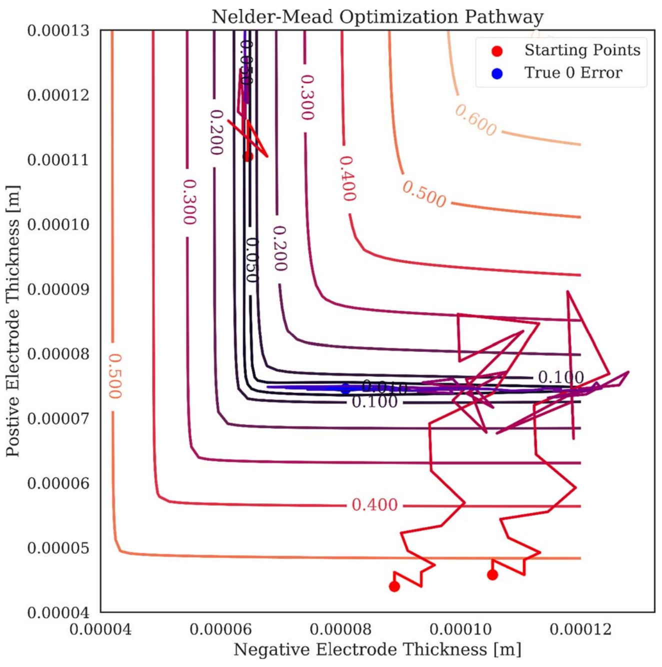


Figure 3. 2D error contour plot demonstrating the optimization pathway based on a set of initial guesses. The Nelder-Mead algorithm fails to achieve an accurate result from one of three starting points due to the nearest error trough, which drives the optimization algorithm away from the true 0 error point. It should be mentioned that Nelder-Mead cannot use constraints or bounds.

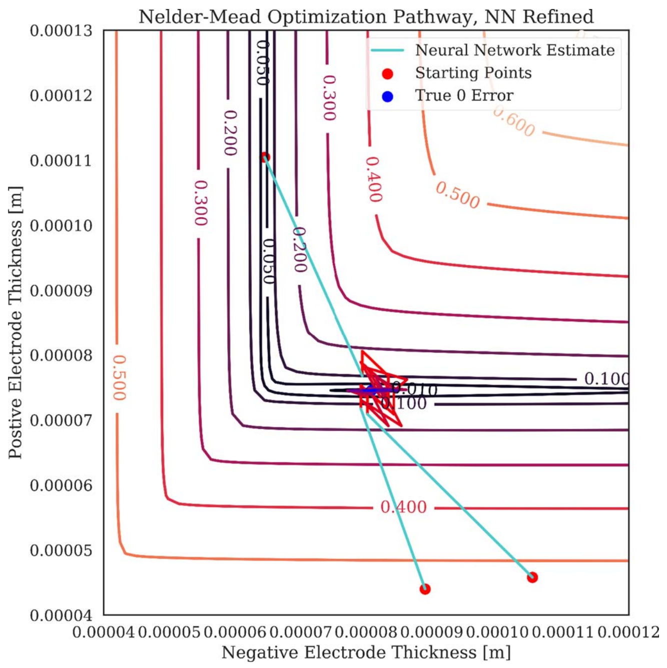


Figure 4. Using the trained neural network as a refinement of the initial guess, all starting points can be made to converge. The changes made by the neural network are highlighted in teal, and the red lines fade to blue as the optimization progresses. In this instance, all optimization algorithms successfully arrive at the true 0 error location.

would change the sensitivity of the parameters arbitrarily, as the end of the curves are thrown away each time. Several example optimization problems are demonstrated with their lines traced on the contour plot, showing the paths the algorithms have taken in their convergence. The demonstrated optimization method here is Nelder-Mead, a simplex-type method implemented in Scientific Python (Scipy), a scientific computing package in Python.[27]

After using Nelder-Mead, a deep neural network was implemented which seeks to act as a problem-specific optimizer, comparing two model outputs and giving the difference in parameters used to create those two curves. The exact formulation of the neural network is discussed in the next section. As seen in Figure 4, when starting at some of the same points as the above optimizer, the neural network gets extremely close to the target value in a single function call, even with only 100 training examples. In this instance, the neural network has demonstrated added value by outperforming the look-up table, meaning that the estimates from the neural network are closer than the nearest training data.

When looking at RMSE as the only metric for difference between two curves, this seems to be an extremely impressive feat – how can all of this information be extracted from a single value? In the case of optimization, the goal is simply to minimize a single error metric, an abstraction from the error between two sets of time series data, in this case. However, during this abstraction, a lot of information is destroyed – in particular, exactly where the two curves differ is completely lost. A similar example can be found in Anscombe's Quartet,[28] which refers to four data sets which have identical descriptive statistics, and yet are qualitatively very different from one another. In Figure 5, the differences in the time series are demonstrated, and it is clear that the curves look very different depending upon which electrode is limiting, the positive electrode or the negative electrode. This information is leveraged in the neural network, but is lost in the naïve optimization. The next section looks to apply this same idea to a higher dimensional space and quantify the benefit that is present when using a neural network to improve a poor initial guess, as is the case with an initial model calibration. Figure 6 demonstrates how the time series data can differ with respect to positive and negative electrode thicknesses in the 2D problem.

Methodology

In this section, the process of using a DNN as a problem-specific optimizer for a 9-dimensional physics-based lithium ion battery model is outlined. The DNN used is typical, with 3 hidden layers comprising 95, 55, and 45 nodes each with 9 nodes on the output dimension, one for each of the model parameters to be estimated. Before being input to the model, each of the parameters was normalized over the range [0,1]. This is separate from sampling, and is done in order to allow the neural network to perform well when guessing large or small numbers. When training a neural network to predict values which vary from one another by several orders of magnitude, it is important to scale the inputs and outputs such that some values are not artificially favored by the loss algorithm due to their scaling. Additionally, when calculating the gradients during training, the line search algorithm assumes all of the inputs are roughly order 1. Another consideration is that the initial values of neural networks are generated under the assumption that the inputs and outputs will be approximately equal to 1. When calculating the loss associated with a misrepresented value, in the case of diffusivities where the target value is $1e^{-14}$, any value near 0 will be considered extremely close to accurate. It is important to note that this normalization is completely separate from the sampling distribution, its only purpose is to force the data to present itself as the neural network engines expect.

The voltages, already fairly close to 1 in value, were left unscaled and unmodified for simplicity. The nonlinearity comes from the eLU activation function,[21] which exponentially approaches -1 below 0 and is equal to the input above 0. This is similar to leaky ReLUs,[29] or LReLUs, which simply have two lines of different slopes that intersect at 0. The point of having a nonzero value below an activation input of 0 is to prevent a phenomenon known as *ReLU die-off*, in which the weights or biases can become trained such that a *neuron* never fires again, as the activation function input is always below 0. This can effect up to 40% of neurons during training. Giving the activation functions a way to recover when an input is below 0 is a way of combatting ReLU die-off, and often results in faster training and better error convergence. Additionally, having an activation function which can output negative numbers reduces the *bias shift* that is inherent to having activation functions which can only produce positive numbers, as with ReLUs, which can slow down learning.[30] Additionally, non-saturating activation functions like eLUs and ReLUs combat vanishing gradients, which can cause significant slowdowns during training when using saturating activations functions like hyperbolic tangents.[31]

The training protocol for this work was adapted from Deepchess,[18] which created a new training set for each epoch by randomly sampling a subset of the data in a comparative manner. This process is leveraged here, where a new training set is generated at each training epoch. For large enough data sets, this can greatly limit overfitting, as the majority of training data given to the neural network is only seen once. The process is described visually in Figure 7, and involves 2 sets of parameter inputs and model outputs, A and B. In the neural network formulation, set A is the current initial guess, while set B is a simulated representation of experimental data, where the numerical model inputs would be unknown, but the desired simulated discharge curve shape is known. The DNN inputs are the scaled parameter set A and the time series difference between the discharge curves generated by parameter sets A and B. The DNN output is a correction factor, calculated as $Output = B - A$. In this way, knowing the correction factor and numerical model inputs A allows for the reconstruction of the desired numerical model inputs, B. During training, the DNN learns to associate differences in the error between the output curves and the current numerical model input parameters with the desired correction factor which will approximate the numerical model input parameters used to generate the second discharge curve.

For the 9-dimensional problem, the input dimension was 1009 – the values of the current scaled parameters and two concatenated discharge

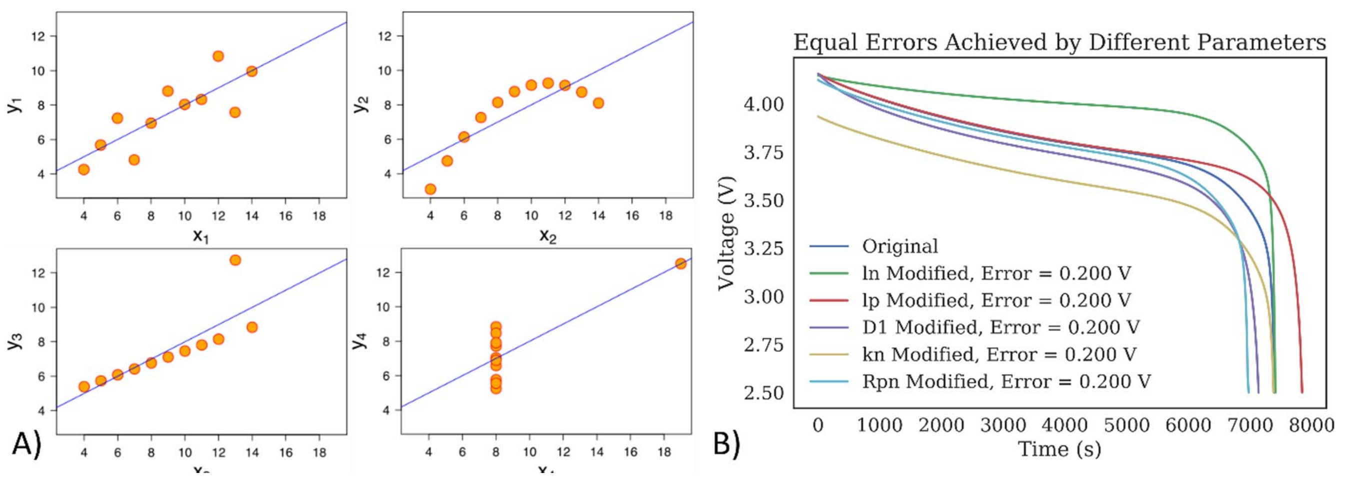


Figure 5. a) Anscombe's Quartet, a set of 4 distributions which have identical second-order descriptive statistics, including line of fit and coefficient of determination for that fit. b) An electrochemical equivalent where the RMSE is equal to 0.200V, but the curves are qualitatively very different. This was achieved by modifying different single parameters to achieve the error.

curves, each of length 500. The size of the training data was varied between 500 and 200,000 samples for the same 9 parameters with the same bounds. For each new training set, the model hyper-parameters remained the same – a batch size of 135, trained for 500 epochs with a test patience of 20, meaning that if test error had not improved in the past 20 epochs, the training ended early, which was used to combat overfitting. The smaller data set models had 50% dropout applied to the final hidden layer in order to fight overfitting, which was present in the smaller data sets. In order to enforce the constant-length input vector requirement of the neural network, the discharge data was scaled with respect to time according to $t = (4000/i_{app})$ uniformly spread over the 500 times steps. Local third-order fits were used to interpolate between the numerical time points. In this text, i_{app} values of 15, 30, 60, and 90 A/m^2 are referred to as 0.5C, 1C, 2C, and 3C, respectively. Additionally, although the neural networks are labeled by the size of the generated data sets, ranging from 500 to 200,000, only 3/4^{ths} of the data were used for training, which is occasionally alluded to in the text. The neural network labeled as needing 200,000 sets of data will have trained on 150,000, while the remainder is held for validation. For all neural networks, the same static test set of 10,000 parameter pairs was used so that the results are directly comparable.

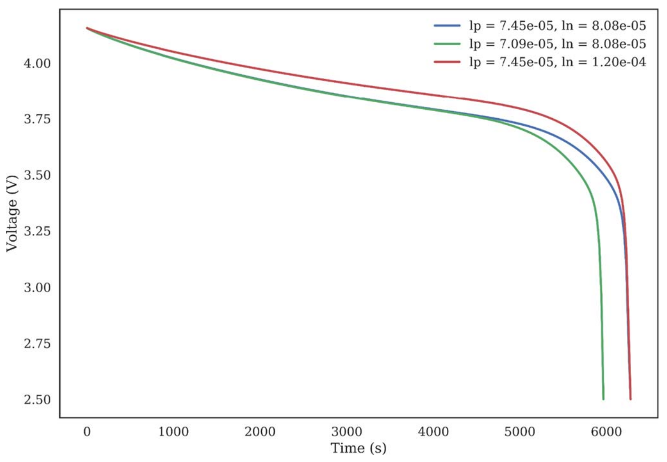
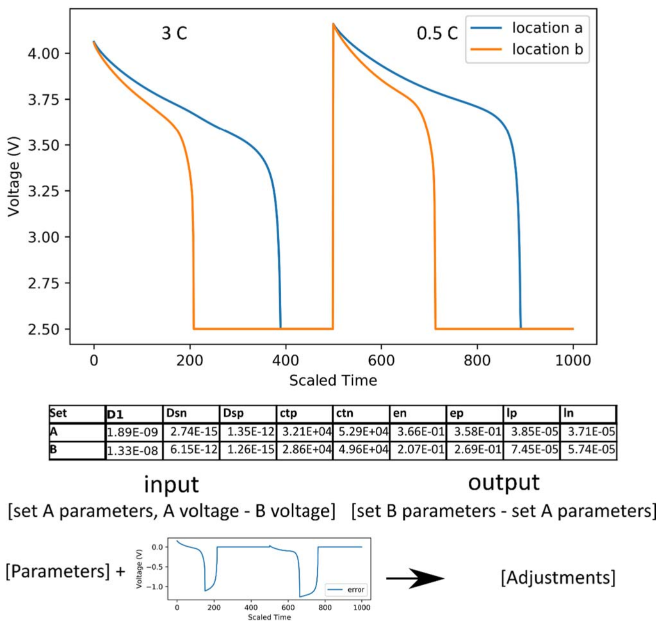


Figure 6. Differences in time series from the 2D example. Changing the positive thickness extends the discharge curve with respect to time, but increasing the negative thickness raises the average voltage without extending the length of the discharge curve.

The purpose of this formulation comes down to two practical considerations – information extraction and physical self-consistency. If a forward model were to be trained on the same data, the model would be responsible for producing physically self-consistent results – for instance, for a constant applied current, the voltage would need to be monotonically decreasing. This requirement is extremely difficult to achieve, especially with finely time-sampled data – the noise will quickly climb above the voltage difference between adjacent points, resulting in an a-physical discharge. In the inverse formulation, where a discharge curve is mapped onto the input parameters via a



Set	D1	Dsn	Dsp	ctp	ctn	en	ep	lp	ln
A	1.89E-09	2.74E-15	1.35E-12	3.21E+04	5.29E+04	3.66E-01	3.58E-01	3.85E-05	3.71E-05
B	1.33E-08	6.15E-12	1.26E-15	2.86E+04	4.96E+04	2.07E-01	2.69E-01	7.45E-05	5.74E-05

Figure 7. A visual representation of the proposed training paradigm. Given some set of model inputs A and another set B, there will be some error between the simulated discharge curves. The goal is to traverse the 9-dimensional space to arrive at the model inputs B. Numerical model inputs A are scaled to be on the set [0,1], concatenated with the error between the discharge curves, and the neural network output is calculated as parameter values A subtracted from parameter values B. In this way, to reconstruct an estimate for parameters B, the NN output must be added to the initial guess A. The flat lines after simulated discharge are artifacts of the constant-length input vector requirements of neural networks, and are not from the model simulation.

Table II. Prediction error at unseen 1C current rate after optimization convergence at 0.5C and 3C rates for various black box optimizers, both with and without neural network initial guess refinement.

		SLSQP		Nelder-Mead		L-BFGS-B		GA	
Optimizer	**Initial Error (V)**	**Function calls**	**RMSE at 1C (V)**	**function calls**	**RMSE at 1C (V)**	**function calls**	**RMSE at 1C (V)**	**function calls**	**RMSE at 1C (V)**
no Neural Network	0.3750	78	0.2236	750	0.0875	418	0.1064		
200k samples	0.0928	46	**0.0727**	577	**0.0218**	489	**0.0283**	17852	0.0470
50k samples	0.0936	71	0.0727	589	0.0277	430	0.0320	4783	0.0501
20k samples	0.0896	51	0.0862	596	0.0264	442	0.0341	2081	0.0511
5k samples	0.2970	111	0.0895	670	0.0533	448	0.0528	765	0.0547
2k samples	0.3273	67	0.1285	648	0.0766	450	0.0751	519	0.0560

neural network, each data point only gets to be used once. This results in significantly poorer guesses for a data set of the same size when compared to this formulation. This is due to the fact that the neural network gets many unique examples where the parameters to estimate are varied – for a data set of 2000 simulations, there are 2000 unique error-correction mappings for every value. This allows the neural network to more intimately learn how each parameter varies as a function of the input error by squaring the effective size of the data set.

There is an important consideration which can be easily illustrated. In the 2D example, an error contour plot was generated which demonstrated the RMSE between curves as a function of their electrode thicknesses with the neural network's mapped corrections superimposed. At first, the performance may look relatively poor, even though it beats the lookup table performance, but it is important to note that this could have been replicated using any of the points and the performance would have been comparable. That is to say that the neural network is not simply learning to navigate one error contour, it is learning to navigate every possible constructed error contour and is able to interpolate between training points in a way that a lookup table of the training data cannot.

9D Application and Analysis

Once the model was trained, it was used as a first function call in a series of optimization schemes. Using three different optimizers, a pre-determined set of 100 A-B pairs was created which mimicked the initial and target discharge curves of a classic experimental fitting optimization problem. Here, however, the concerns about the model being able to adequately match the experimental data are removed, meaning that any measurable error between the converged value and the target curves are due to optimizer limitations and not due to an inability of the model to describe the experiment.

As shown in Table III, passing the initial simulation through the neural network optimizer can drastically reduce the error at convergence, improving the final converged error by 4-fold. The optimization methods tested here include Sequential Least Squares Programming (SLSQP),[32] Nelder-Mead,[33] the quasi-newton, low-memory variant of the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) algorithm (L-BFGS-B),[34] and a genetic algorithm (GA).[35] Each of these optimization methods works fundamentally differently, and they were selected in order to adequately cover the available optimizers. All of the methods are implemented by Scipy[27] in Python. SLSQP and L-BFGS-B are bounded optimizers, while Nelder-Mead cannot handle constraints or bounds.

The first column of Table II shows the average initial error between points A and B, which equals 375 mV. After passing these relatively poor initial guesses through the neural networks once, the errors are demonstrably improved. Interestingly, although the 20k sample neural network creates guesses with a lower initial error on the unseen data, the converged results after optimization are consistently worse. This could be a coincidence of sampling, and can likely be attributed to the fact that the parameters given to the DNN do not have identical sensitivity in the physical model. Looking at the values of the relative error of the parameters, as in Table III, the models stack according to intuition, with the initial guesses being extremely wrong, and the neural networks performing significantly better, ordered in performance by the size of their training data. Note that these massive parameter errors are possible because the diffusivities vary by three orders of magnitude. These values were calculated on the parameters after they had been descaled, and as such, the effects of the accuracy in the diffusivities are likely dominant due to their massive variance relative to the other values.

The genetic algorithm is implemented in the differential evolution function through Scipy[35] and the number of generations and size of the generations were varied to reflect the training size divided by the number of iterations, leaving between 20,000 and 200 function evaluations per optimization. Unfortunately, the genetic algorithms could not be seeded with any values, which means that the neural network outputs could not be used. The limitation on the number of viable function calls was the compromise for restricting information for the genetic algorithm. In this instance, *polishing*, or finishing the genetic algorithm with a call of L-BFGS-B, was set to False. All other defaults were left, other than the maximum number of iterations and the population size. The number of iterations and population size were not optimized for the best performance of the genetic algorithm. For SLSQP, in order to convince the optimizer that the initial guess was not a convergent state, the step size used for the numerical approximation of the jacobian was changed from 10^{-8} to 10^{-2}. Everything else was left as default.

In general, it is apparent that refining a relatively poor initial guess using the neural network optimizer can improve convergence and reduce the error, which was calculated at unseen data. A typical optimization pathway is demonstrated below in Figure 8, which shows the error between the optimizer and the target data as a function of the number of function evaluations. The method used below was Nelder-Mead, an algorithm which is not extremely efficient in terms of the average number of steps needed for convergence, but it is relatively robust to local minima and generally produces the best results with this model, as seen in Table II. It is apparent from Figure 8 below that improving an initially relatively poor guess with the neural network transports the optimizer across a highly nonconvex space, allowing it to converge to a much more accurate local minimum in fewer function calls than without using the neural network optimizer. Interestingly, there are several points where the unrefined error is lower than the

Table III. Test set loss and mean relative absolute error of descaled parameters.

Method	Mean Relative Absolute Error of Parameters (%)	Test Set Error
Initial	2958	
200k NN	56	.0324
50k NN	58	.0391
20k NN	79	.0393
5k NN	143	.0634
2k NN	343	.0707

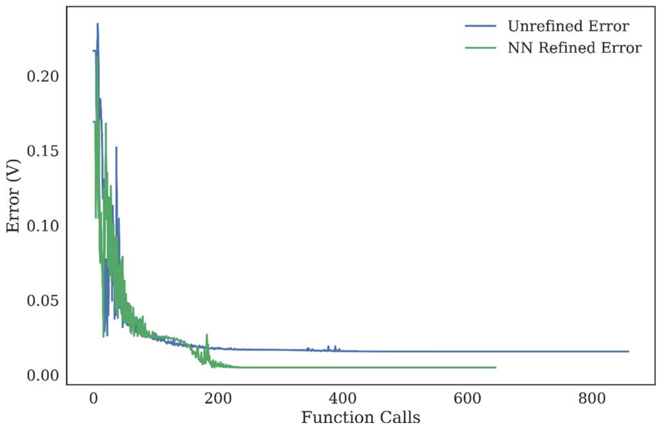


Figure 8. Typical optimization pathway for Nelder-Mead, where the current estimate error is displayed vs the function call number. Starting from the same initial guess with an error of 220mV, the guess refined by the neural network converges to a significantly lower error than the unchanged guess.

refined error, which hints at the relative number of local minima in the optimization space. Considering Figure 5b, it is apparent that given the number of different ways the parameters can change the shape of the discharge curve, it is not surprising that there are multiple avenues for the reduction of error, even if the result is not the globally optimal result.

While the average case is improved, it is important to note that the cost of having a small neural network which is also generalized implies an accuracy tradeoff – if the initial guess is very good, the neural network may suggest changes which make the guess worse, in which case, the output can be ignored and the total opportunity cost of attempting to improve an initial guess with the neural network was only a single extra function call. Additionally, while all of the training data is physically meaningful, there is no guarantee that the per-parameter corrections given by the neural network optimizer will result in physically meaningful parameters. Although the training data bounds are sensible, there is no guarantee that an output from the neural network will fall within these bounds, and no meaningful way to predict when it will happen. Aside from changing the initial guess and estimating with the neural network again, there is no way to quickly repair an a-physical estimate. For example, the neural networks trained on smaller data sets have recommended negative diffusivities. In this instance, the guess was thrown out.

Lookup table performance is examined in the next section, which directly compares the neural networks with genetic algorithms in terms of using best-of-set sampling for generating initial guesses. The neural network size selected was chosen to compromise between simplicity and accuracy – a smaller network will prefer generalizability to over-fitting, but with an extremely large data set, better test error may be achieved with a larger network despite the tendency to over-fit. There are additional considerations as well, such as size on disk and time to compute. A look-up table is O(n), and as such takes around 2.5 seconds for the training set of 150,000 simulations, stored in a Numpy array, loaded into memory. When stored in an HDF5 array using H5Py,[36] the array does not need to be pulled into memory, but can be loaded one chunk at a time. While this is extremely beneficial for large datasets, it is also considerably slower, taking 10 minutes to look through the table once. Additionally, the compressed size of the data on disk is 1.04 GB. Time to evaluate the neural network is roughly 1e-6 seconds, and the size on disk is 1.2 MB – significantly faster and smaller than keeping the compressed data. While modern hardware can handle these larger files easily, if model estimation is attempted on cheaper or more limited hardware, these may not be acceptable.

While this work focuses on using this formulation to fit discharge curves of simulated lithium ion batteries, any multi-objective optimization which is compressed to a single objective for an optimizer could stand to benefit from framing a neural network as a single-step optimizer rather than compress the multiple objectives into a single value and pass that to a traditional black box optimizer. This technique does not eliminate the value of traditional optimizers, however; the best result comes from using the neural network to refine a poor guess, as is often the case when starting an optimization problem. The neural network can never achieve arbitrary precision the same way a traditional optimization algorithm can. The goal of the neural network optimizer is to traverse the highly non-convex space between an initial guess and the target parameters, and give an easier optimization problem to the traditional, theoretically proven optimization methods.

9D Application - Genetic Algorithm Comparison

While the above method is useful for analyzing the performance for navigating in a 9-dimensional parameter space and trying to get from some point A to some point B, those who wish to arrive at the best result for an optimization problem often do not place much value on the initial guess. In these instances, other methods may first be used to explore the parameter space, and the best result from that will be used as an initial guess for the new optimization problem using a more traditional optimizer.

There is an analog for this deep neural network optimization method as well, where after the neural network is trained, instead of starting from some unseen data point, a random sampling of training or test points are fed in as initial guesses, where the target curves remain the original targets. This has the added advantage of not requiring simulations to get a parameter guess, only to check the error of the guess. This technique leverages the idea that the neural network is only hyper-locally accurate, and that an accurate guess at the parameters cannot be guaranteed from any point in the 9-dimensional space. However, by sampling several points, it is possible to end up with an extremely performant guess for the cost of a few simulations.

To understand how this method is in direct competition with a genetic algorithm, that approach must first be explained. For each generation, a series of randomly generated points are created and evaluated. At the end of each generation, the best few results are kept and either randomly permuted, or combined with other results in order to create the next generation, which is then evaluated. This technique is popular in the literature, but it tends to be very function-call inefficient, converging only after many evaluations.

For this optimization, the formulation is equivalent – sets of input parameters are either randomly generated or are randomly selected from a list of pre-generated, unseen parameters. These are then offered to the neural network, along with the associated error between the discharge curves, and a refined guess is generated by the neural network. The value of this guess is very difficult to determine without examining the result, and can be significantly better or significantly worse depending on many factors, including the sampling density of the parameter space, the size of the parameter space, and the initialized weights of the neural network. Although it may take several function calls to end up with a good guess, this guess is often very accurate, and for very sparse sampling it can compete with the lookup table performance.

Using the same data as the previous section, a new static set of 100 final target values was created by randomly sampling from the test data. There are two main differences between this analysis and the previous analysis – rather than interpolating the currents, a new set of neural networks were trained on discharges at 0.5C and 1C, and the reported RMSE values are the summed result of calculation at these two currents. From the results in Table IV, it is clear that the neural networks perform significantly better than the genetic algorithm per function call, and that 100 random, unseen samples is sufficient to approximate the lookup table performance after passing the guesses through the neural network. While the lookup table seems to outperform the neural network, the converged results examined in Table V reinforce the idea that the root mean squared error metric does not tell the whole story, and the neural networks outperform the lookup table at low sampling densities.

Table IV. Average root mean squared errors of the best-of-sampling results from each initial guess generation technique, fitting simulations at 0.5C and 1C. The error reported here is the sum of RMSE of voltage at 0.5C and 1C.

NN training size	NN test error	From initial (V)	10 samples (V)	20 samples (V)	50 samples (V)	100 samples (V)	Lookup Table (V)
200k	0.0412	0.0895	0.0482	0.0285	0.0150	0.0106	0.0072
50k	0.0419	0.0960	0.0704	0.0460	0.0305	0.0226	0.0083
20k	0.0423	0.0897	0.0408	0.0285	0.0190	0.0132	0.0128
5k	0.0526	0.1129	0.0723	0.0500	0.0260	0.0193	0.0205
2k	0.0644	0.1275	0.0918	0.0646	0.0371	0.0283	0.0292
500	0.0839	0.1960	0.1605	0.1108	0.0646	0.0494	0.0485
		18 samples	81 samples	144 samples	990 samples	2079 samples	
GA		0.2044	0.0784	0.0568	0.0275	0.0256	

In Figure 9 below, the resulting RMSE as a function of number of random guesses is examined for each of the neural networks and compared with a genetic algorithm. The neural network errors were sampled at 1, 10, 20, 50, and 100 random inputs, and the genetic algorithm was sampled as closely to 20 and 100 samples as math would allow, as the number of function calls scales as $len(x) * (maxiter + 1) * popsize$. Since the length of x is 9 in this instance, it was not possible to perfectly hit the desired sampling values.

Looking at the results in Table IV, some clear patterns emerge. The error of the guesses from the neural network is extremely high for the more coarsely sampled training sets, resulting in very poor error for the first function call, all of which were taken from the same initial guesses. After this initial function call, however, the guess points were randomly sampled from unseen data and the accuracy of the improved points was examined. The best results from each sample range are kept.

In this work, neural networks are fairly small for this size of input data, where the input dimension is 1009, but the largest internal node size is only 95. This was done in order to force the neural network to generalize more aggressively, which can often improve the performance on unseen data when compared to a larger network which can memorize the dataset, but tends to over fit. This is classically known as the bias-variance tradeoff.[37] The size of the neural network was kept constant across sampling rates, in order to increase interpretability, meaning the network size is likely too large for the very coarse sampling and too small for the very fine sampling, and changing the neural network hidden layer dimensions to suit the sampling would result in better performance. It is worth mentioning that genetic algorithms are iterative in nature, meaning that the population generation and evaluation is embarrassingly parallelizable, but the number of iterations is inherently serial. For the neural networks, no action is serial, so the entire sampling is embarrassingly parallel, which can drastically decrease the time to an accurate estimate for long function calls.

After these initial guesses were collected, Nelder-Mead was used to polish the optimization result and the number of function evaluations needed to accomplish this task were recorded, along with the average final value. The parameter-level tolerance for convergence was set to 1e-6, and the maximum number of iterations was set sufficiently high to allow convergence. These results are compared to the output values from the genetic algorithm and are compared in Table V. It is important to note that the initial errors between the static starting points and ending points was 0.9004V, and Nelder-Mead converged at an average RMSE of 0.0327 V when starting the optimization from that point. By refining the initial guess with a neural network, it was possible to drastically reduce this error to 0.0032 V, representing a 10-fold reduction in error. However, by forgoing this constraint entirely, it was possible to get an even lower 0.00137 V error after getting a very close initial guess by randomly sampling the neural network.

This performance offers a significant improvement over the converged error after polishing the output of the genetic algorithm, which had errors which were comparable in value to the neural network sampling, but which resulted in significantly worse convergent error. For example, by generating an initial guess using a genetic algorithm over the same bounds used for the generation of the training data, limiting the number of function calls to 990 spread across a population size of 11 and 9 iterations, an initial error of 8.3mV can be achieved. A comparable initial guess error can be found either using a neural network trained on 20,000 simulations and sampled 50 times, or a neural network trained on 5,000 simulations and sampled 100 times. However, after optimization, the initial guess from the genetic algorithm averages to 2.6mV error, while the initial guesses provided by the neural network converge to 0.48mV and 0.6mV error, respectively – a 5-fold improvement over the existing technique, despite comparable initial errors.

These results are summarized in Figure 10, which serves to showcase the idea that RMSE is not a sufficient metric when considering the difficulty an optimization algorithm will face when attempting to minimize the error from a given set of initial conditions. If a vertical

Table V. Final converged root mean squared errors as a function of number of samples, neural network training data size, and genetic algorithm function evaluations. The error is the sum of RMSE for 1C and 0.5C discharges.

NN Training size	from initial		10 samples		20 samples		50 samples		100 samples		Lookup Table	
	num calls	final error (v)	num calls	final error (v)	num calls	final error (v)	num calls	final error (v)	num calls	final error (v)	num calls	final error (v)
200k	705	0.003205521	674	0.002464	662	0.001637	659	0.001533	654	0.00137	1752	0.001129
50k	1137	0.004354459	1745	0.002323	1710	0.002444	640	0.002879	643	0.001964	663	0.001116
20k	1822	0.003040758	685	0.001741	695	0.001687	652	0.001493	646	0.001554	715	0.002721
5k	736	0.006148043	1777	0.004997	3902	0.003494	682	0.002625	671	0.002265	2863	0.004176
2k	1841	0.008858207	765	0.006464	758	0.006183	2214	0.003428	2218	0.002825	756	0.007579
500	787	0.013215034	1873	0.014534	784	0.011733	765	0.010926	772	0.011221	797	0.01185
GA	18 evaluations		81 evaluations		144 evaluations		990 evaluations		2079 evaluations			
	num calls	final error	num calls	final error	num calls	final error	num calls	final error	num calls	final error		
	2867	0.03054	1790	0.02772	713	0.02483	2792	0.0184	657	0.0191		

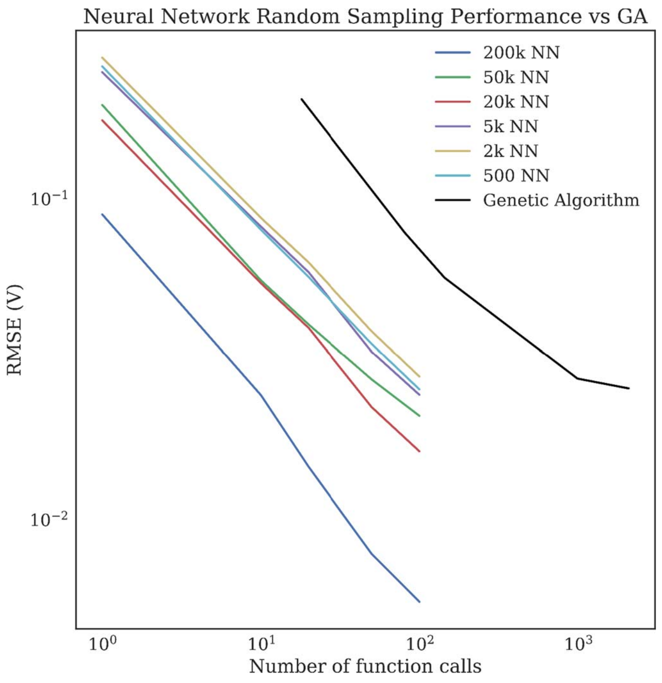


Figure 9. The root mean squared error between the best estimate from the neural networks and a genetic algorithm as a number of function calls. The neural networks are all similar, with errors decreasing with increasing training size. The genetic algorithm shares a qualitatively similar relationship, but the error begins to plateau by 1000 function calls, while the error is still higher than that of the neural networks.

line were drawn on Figure 10, it would indicate an equal initial error. Plotting intuition on this graph would likely look like a line with a 45-degree slope, indicating that any guess with a lower initial error would result in a converged result with lower error. Instead, it is clear

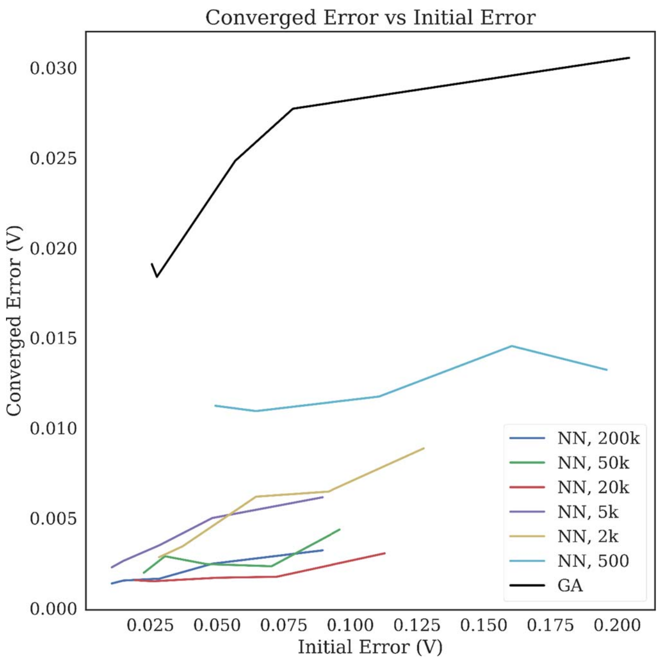


Figure 10. The converged error vs the initial error, grouped by method of guess generation. The neural networks clearly perform significantly better than the genetic algorithm, despite having many points which have comparable initial errors to the genetic algorithm.

that guesses generated by the neural networks and those generated by the genetic algorithm perform significantly differently, even when the value of the initial error is identical. This suggests that there is something which the neural networks are doing extremely well which the genetic algorithm is doing poorly, and that the guesses generated by the neural networks result in much easier optimization problems than those generated by the genetic algorithm. Even in the data-starved or data-comparable conditions of 500 and 2000 samples for training, the neural network outperforms the best-performing genetic algorithm.

Although it is popular to use a genetic algorithm to get an initial guess followed by a traditional optimization technique, genetic algorithms are extremely inefficient in terms of performance per function call. The neural networks trained on Sobol-sampled data outperform the genetic algorithm, even with small amounts of data – the smallest neural network trained on only 500 sets of generated data and sampled 100 times matches the genetic algorithm performance for 600 total function calls, compared to 990 for the genetic algorithm. This means that the neural network will outperform the genetic algorithm, even when starting from scratch. In addition, neural networks have the added benefits of being entirely embarrassingly parallelizable and are inherently reusable for new optimization problems. The ease of deployment and speed of neural networks make them a convenient way to compress the information from a large number of generated data points into a compact and easily accessible size, trading off some linear algebra for gigabytes of compressed data.

This discrepancy between the success of the optimization algorithm and the initial errors of the guesses is extremely interesting and serves to both demonstrate the inadequacy of a single value to represent the deviation between two discharge curves and to emphasize the importance of building parameter sensitivity into an initial guess generation method. For the current implementation of the genetic algorithm, no parameter scaling was done, which likely led to oversampling of large values in the range of diffusivities, which vary by three orders of magnitude. However, the lookup table results also feature a large error on the diffusivity-related parameters, which serve to demonstrate the extremely low sensitivity to these parameters when lithium diffusion is not the limiting factor that shapes the discharge curve.

An additional analysis was done using the lookup table results for the training sets of each of the neural networks. While randomly sampling 100 sets of parameter values was sufficient to approximate the error performance of the lookup table for each neural network, an interesting trend results from optimizing based on those recommended values. For coarser sampling, the RMSE of the converged values is improved by two fold compared to using the best fit from the training data, as shown in Table VI. Examining the error between the target parameters and the estimated parameters reveals that the neural networks are significantly better at estimating the parameters than using a lookup table, shown in Figure 11. A clear pattern emerges, wherein the optimization results based on the neural network's output outperform the training data until the sampling becomes so fine that the small neural network size tends to limit the accuracy of the model outputs, and the resulting errors well below one millivolt begin to converge. While this problem was done with 9 dimensions, the full dimensionality of the P2D model is 24, which would require significantly more samples to adequately explore.

It is clear that the RMSE between two curves does not fully predict the ease with which an optimizer will converge to an accurate solution. Two different problems have been analyzed, one of which was 2-dimensional for the purposes of visualizing the process, and the other of which was 9-dimensional for the purposes of exploring a practical set of optimization problems in higher dimensions. Potential future work would include extending this analysis to higher dimensions, using battery simulations with differing sensitivities to the parameters, or perhaps combining these approaches with Electrochemical Impedance Spectroscopy measurements for increased sensitivity to other parameters. Additionally, the same underlying assumptions used when generating the data are applied to the applicability of the model. For example, to assume that the parameter space is uniformly varying across a parameter range is likely false, in particular with

Table VI. The RMSE of the previous converged results at the unseen condition of a 2C discharge. This represents an extrapolation in terms of current values, as 2C is higher than the 0.5C and 1C used for curve fitting. The NN and lookup table significantly outperform the GA, and the neural network outperforms the lookup table, especially when sampling is coarse.

NN Training size	Initial (V)	converged (V)	GA	initial (V)	converged (V)	Lookup Table	initial (V)	converged (V)
200k	0.01204	0.003065	**2079**	0.04358	0.03711	**200k**	0.0111	0.003117
50k	0.01841	0.003403	**990**	0.0438	0.03373	**50k**	0.01102	0.003569
20k	0.01462	0.003453	**144**	0.05362	0.03681	**20k**	0.01594	0.00493
5k	0.01963	0.004788	**81**	0.06227	0.03894	**5k**	0.02299	0.00648
2k	0.02536	0.0048644	**18**	0.1084	0.03911	**2k**	0.02916	0.01083
500	0.04137	0.01804				**500**	0.04202	0.01611

electrode thickness – it is likely closer to a bimodal distribution, as some cells are optimized for energy and others for power. Sampling the types of batteries to be fit and using the distributions of parameters can increase the chances of success when calibrating the model to experimental batteries.

Conclusions

In this work, a deep neural network was used both to refine a poor initial guess and to provide a new initial guess from random sampling in the parameter space. For the execution cost of one additional function call, it is reasonable to improve the final converged error on unseen data by 100-fold when compared with random model parameterization, often with fewer total function calls. It should be noted that this performance is on an exceptionally poor guess, indicating the difficulty optimizers have with this model. However, by randomly generating data and feeding these points into the neural network, it is possible to get an extremely good fit for under 100 function calls, which improves final converged error by 5-fold compared to generating an initial guess using a genetic algorithm, and improves the error by 10-fold when evaluating model performance at higher currents. This framework is generally applicable to any optimization problem, but it is much more reliable when the output of the function to be optimized is a time series rather than a single number. Additionally, the outputs of the neural network after 100 function calls outperform the lookup table of the training data, indicating value added by the ability to interpolate between data points in high dimensional space, while taking significantly less space on disk than the training data. The deep neural network can easily be implemented in any language which has matrix mathematics, as was done with Numpy in Python. In this instance, the neural network acts as a much more efficient encoding of the data, replacing a lookup table with a few hundred element-wise operations and replacing a gigabyte of data with a megabyte of weights and biases.

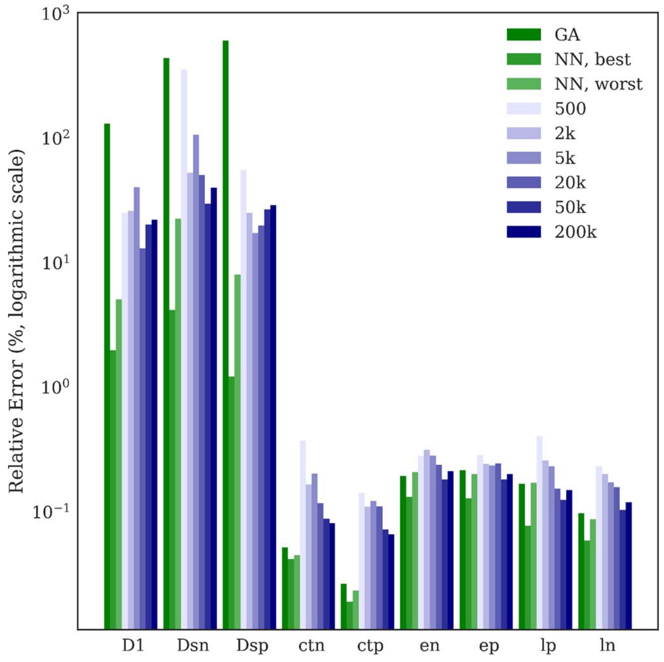


Figure 11. Relative per-parameter error for best GA, neural network, and training data. The neural network clearly offers the best performance, demonstrating significantly better performance than a lookup table of the training data.

The optimization formulation of the neural network leverages the increased informational content of a difference between time series over an abstracted summary value, like root mean squared error, which is required by many single-objective optimizers. Additionally, the ability of neural networks to take advantage of data shuffling techniques allows the algorithm to efficiently combat overfitting for only minimal computational overhead during training. This formulation also allows the neural network to extract the maximum amount of information from the generated data when compared to the inverse formulation, in which each discharge curve and input parameter pair is only seen once. The source code for this work can be found at www.github.com/nealde/EChemFromChess, along with examples and code for generating some of the images.

Acknowledgments

The authors thank the Clean Energy Institute (CEI) at the University of Washington (UW) and the Data Intensive Research Enabling Clean Technologies (DIRECT), a UW CEI program funded by the National Science Foundation (NSF), in addition to the Washington Research Foundation (WRF) for their monetary support of this work. The battery modeling work has been supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Vehicle Technologies of the U. S. Department of Energy through the Advanced Battery Material Research (BMR) Program (Battery500 Consortium).

ORCID

Suryanarayana Kolluri https://orcid.org/0000-0003-2731-7107
Venkat R. Subramanian https://orcid.org/0000-0002-2092-9744

References

1. M. Guo, G. Sikha, and R. E. White, *J. Electrochem. Soc.*, **158**, A122 (2011).
2. P. W. C. Northrop et al., *J. Electrochem. Soc.*, **162**, A940 (2015).
3. P. W. C. Northrop, V. Ramadesigan, S. De, and V. R. Subramanian, *J. Electrochem. Soc.*, **158**, A1461 (2011).
4. J. Newman and W. Tiedemann, *AIChE J.*, **21**, 25 (1975).
5. M. Doyle, T. F. Fuller, and J. Newman, *J. Electrochem. Soc.*, **140**, 1526 (1993).
6. M. Pathak, D. Sonawane, S. Santhanagopalan, R. D. Braatz, and V. R. Subramanian, *ECS Trans.*, **75**, 51 (2017).
7. H. Perez, N. Shahmohammadhamedani, and S. Moura, *IEEEASME Trans. Mechatron.*, **20**, 1511 (2015).
8. S. Tansley and K. M. Tolle, *The Fourth Paradigm: Data-intensive Scientific Discovery*, p. 292, Microsoft Research, (2009), https://www.microsoft.com/en-us/research/publication/fourth-paradigm-data-intensive-scientific-discovery/.
9. N. Dawson-Elli, S. B. Lee, M. Pathak, K. Mitra, and V. R. Subramanian, *J. Electrochem. Soc.*, **165**, A1 (2018).
10. S. S. Miriyala, V. R. Subramanian, and K. Mitra, *Eur. J. Oper. Res.*, **264**, 294 (2018).
11. P. D. Pantula, S. S. Miriyala, and K. Mitra, *Mater. Manuf. Process.*, **32**, 1162 (2017).

12. S. S. Miriyala, P. Mittal, S. Majumdar, and K. Mitra, *Chem. Eng. Sci.*, **140**, 44 (2016).
13. W. Zhang, K. Itoh, J. Tanida, and Y. Ichioka, *Appl. Opt.*, **29**, 4790 (1990).
14. Y. LeCun and Y. Bengio, in M. A. and Arbib, Editor, p. 255, MIT Press, Cambridge, MA, USA (1998) http://dl.acm.org/citation.cfm?id = 303568.303704.
15. A. van den Oord, S. Dieleman, and B. Schrauwen, in *Advances in Neural Information Processing Systems 26*, C. J. C. , Burges, L. , Bottou, M. , Welling, Z. , Ghahramani, K. Q. , and Weinberger, Editors, p. 2643, Curran Associates, Inc. (2013) http://papers.nips.cc/paper/5004-deep-content-based-music-recommendation.pdf.
16. V. K. Singh et al., *ArXiv180901687 Cs* (2018) http://arxiv.org/abs/1809.01687.
17. R. Collobert and J. Weston, in *Proceedings of the 25th International Conference on Machine Learning, ICML '08.*, p. 160, ACM, New York, NY, USA (2008) http://doi.acm.org/10.1145/1390156.1390177.
18. E. David, N. S. Netanyahu, and L. Wolf, *ArXiv171109667 Cs Stat*, **9887**, 88 (2016).
19. A. Karpatne, W. Watkins, J. Read, and V. Kumar, *ArXiv171011431 Phys. Stat* (2017) http://arxiv.org/abs/1710.11431.
20. B. Raj, *Medium* (2018) https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced.
21. D.-A. Clevert, T. Unterthiner, and S. Hochreiter, *ArXiv151107289 Cs* (2015) http://arxiv.org/abs/1511.07289.
22. V. Nair and G. E. Hinton, in *Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10.*, p. 807, Omnipress, USA (2010) http://dl.acm.org/citation.cfm?id = 3104322.3104425.
23. G. B. Goh, N. O. Hodas, and A. Vishnu, *ArXiv170104503 Phys. Stat* (2017) http://arxiv.org/abs/1701.04503.
24. D. P. Kingma and J. Ba, *ArXiv14126980 Cs* (2014) http://arxiv.org/abs/1412.6980.
25. I. M. Sobol′, *Math. Comput. Simul.*, **55**, 271 (2001).
26. A. Saltelli et al., *Comput. Phys. Commun.*, **181**, 259 (2010).
27. J. E, O. E, and P. P, (2001) http://www.scipy.org/.
28. F. J. Anscombe, *Am. Stat.*, **27**, 17 (1973).
29. A. L. Maas, A. Y. Hannun, and A. Y. Ng, 6.
30. Y. L. Cun, I. Kanter, and S. A. Solla, *Phys. Rev. Lett.*, **66**, 2396 (1991).
31. J. Lin and L. Rosasco, in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Editors, p. 4556, Curran Associates, Inc. (2016) http://papers.nips.cc/paper/6213-optimal-learning-for-multi-pass-stochastic-gradient-methods.pdf.
32. D. Kraft, and D. F. V. für L. R. (DFVLR) Institut für Dynamik der Flugsysteme, *A software package for sequential quadratic programming*, DFVLR, Braunschweig, (1988).
33. F. Gao and L. Han, *Comput. Optim. Appl.*, **51**, 259 (2012).
34. M. Sarrafzadeh, *ACM SIGDA Newsl.*, **20**, 91 (1990).
35. R. Storn and K. Price, *J. Glob. Optim.*, **11**, 341 (1997).
36. http://www.h5py.org/.
37. C. Sammut and G. I. Webb, Eds., in *Encyclopedia of Machine Learning*, p. 100, Springer US, Boston, MA (2010) https://doi.org/10.1007/978-0-387-30164-8_74.