Learning Parametric Convex Functions

Maximilian Schaller Alberto Bemporad Stephen Boyd

June 4, 2025

Abstract

A parametrized convex function depends on a variable and a parameter, and is convex in the variable for any valid value of the parameter. Such functions can be used to specify parametrized convex optimization problems, *i.e.*, a convex optimization family, in domain specific languages for convex optimization. In this paper we address the problem of fitting a parametrized convex function that is compatible with disciplined programming, to some given data. This allows us to fit a function arising in a convex optimization formulation directly to observed or simulated data. We demonstrate our open-source implementation on several examples, ranging from illustrative to practical.

Contents

1 Introduction

1.1 Parametrized convex functions

A *parametrized convex function* (PCF) f has the form

$$f : \mathbf{R}^n \times \Theta \to \mathbf{R}^d,$$

where $\Theta \subseteq \mathbf{R}^p$. To be a PCF, f must be continuous in θ, and for each $i = 1, \ldots, d$, $f_i(x, \theta)$ is convex in x for any $\theta \in \Theta$. We refer to the first argument x of the PCF f as the *variable*, and the second argument θ as the *parameter*. When $d = 1$, we refer to f as a scalar PCF.

1.2 Disciplined convex programming

The terms variable and parameter in a PCF are taken from disciplined convex programming (DCP), a method for expressing a PCF as an expression in a domain specific language (DSL) constructed from variables, constants, parameters, and a small library of functions called atoms [AVDB18, DB16]. In DCP, the expression must be constructed in a specific way that corresponds to a composition rule that establishes convexity of the function with respect to the variable, for any valid parameter.

Functions expressed in DCP form can be used to form a parametrized convex optimization problem or convex optimization family. When the parameters are given specific numerical values, we obtain a problem instance, which can be solved by automatically transforming the problem instance to a canonical form, solving the canonical form, and then retrieving the solution of the original problem instance from the solution of the canonicalized problem instance. Examples of DSLs that leverage DCP for modeling convex optimization problems (and support parameters) include CVXPY [DB16] (in Python), CVXR [FNB20] (in R), Convex.jl [UMZ+14] and JuMP [DHL17] (in Julia). Precursors that do not handle parameters include CVX [GB14] and YALMIP [Lö4] (in Matlab).

1.3 Examples

PCFs arise in many applications, including control, machine learning, resource allocation, and finance, to name just a few. We describe below a few typical ones, some specific and some more generic. In our examples we assume that f is a scalar PCF.

Fuel use map. Here $f(x, \theta)$ gives the instantaneous fuel use rate. The variable x might correspond to thrust or power output, variables that typically appear in an optimization problem; the parameter θ contains additional parameters that affect the fuel use, such as temperature. When modeling the total fuel use of a set of actuators on a moving vehicle, x can represent the net force and torque on the vehicle, and θ can contain the vehicle orientation, which would affect which actuators are used to obtain the required force and torque.

Battery aging model. Here $f(x, \theta)$ gives the rate of aging of a battery, *i.e.*, reduction of capacity due to use, relative to the initial battery capacity. The variable x corresponds to the battery charge/discharge rate, which might appear in an optimization problem. The parameter θ contains other quantities that affect aging of the battery, such as its temperature.

Convex optimization control policy (COCP). A control policy maps the context, *i.e.*, what is known at a given time, such as the state of a dynamic system and other measurable quantities, into an action denoted $u \in \mathcal{U} \subseteq \mathbf{R}^r$. In a convex optimization control policy (COCP) [BAB20] the action u is found as the solution of a convex optimization problem that is parametrized by the context. As a concrete example, $f(x, \theta)$ gives the cost of the action u, combined with the long term cost of the next state, which depends on the action. The parameter θ contains the context observed when computing the control action [RMD+17, KC16, GPM89].

Resource allocation. Here $f(x, \theta)$ gives the cost (or negative utility) of providing resources specified by the vector $x \in \mathbf{R}^n$ to n agents. The parameter θ might contain information like time of the year, month, or day.

Financial portfolio construction. The goal of financial portfolio construction is to find a portfolio of financial assets that maximizes the expected return of the portfolio while limiting the investment risk. The function $f(x, \theta)$ might represent a combination of expected return and risk as a function of x, which describes the portfolio. The parameter θ provides information about market conditions, or current forecasts.

1.4 Learning a parametrized convex function

This paper concerns learning a PCF f from data, *i.e.*, fitting a PCF to data,

$$(x^k, \theta^k) \in \mathbf{R}^n \times \Theta, \quad y^k \in \mathbf{R}^d, \quad k = 1, \ldots, N. \tag{1}$$

The PCF f is specified by its architecture and a choice of *model weights*, which we denote $w \in \mathbf{R}^q$. We require our approximation f to be DCP expressible, which implies that it can be used to construct parametrized convex problems. This fitting method allows us to learn a DCP expression directly from observed (or generated) data.

Fitting method. We use a standard regularized loss method to choose a set of model weights. Let $\ell : \mathbf{R}^q \times \mathbf{R}^n \times \Theta \times \mathbf{R}^d \to \mathbf{R}$ denote a loss function, and $r : \mathbf{R}^q \to \mathbf{R}$ a regularizer function. The loss function is used to judge how well our approximation fits the data, and the regularization function is meant to penalize complexity. We choose w to (approximately) minimize the regularized average loss,

$$\frac{1}{N} \sum_{k=1}^{N} \ell(w; x^k, \theta^k, y^k) + \lambda r(w), \tag{2}$$

where $\lambda \geq 0$ is a hyper-parameter that scales the regularization. In this context we refer to the data as training data, since it is used to train or learn the PCF.

As in all data fitting methods we judge a candidate PCF by its accuracy on unseen data. (The loss used to evaluate the model can differ from the loss ℓ used to train the model.) To find a PCF that performs well on unseen data we use the standard technique of cross-validation. We partition the original data into K groups of approximately equal size, called folds, and for each fold we train the model weights on the data not including that fold, and evaluate the average loss on the data in that fold. We average the losses for the K folds to obtain an overall fitting metric. We choose the architecture and the regularization hyper-parameter λ so as to minimize the overall fitting metric. Once the architecture and hyper-parameter are chosen, we fit the PCF using all the data.

Special cases. The PCF learning problem, *i.e.*, minimizing (1), reduces to well known problems in special cases. When $n = 0$, *i.e.*, there is no variable, it reduces to the very general problem of fitting a continuous function from Θ into $\mathbf{R}^d$. When $p = 0$ (or Θ is a singleton), there is (effectively) no parameter, and the problem reduces to fitting a convex function to some given data. Several architectures have been developed for this task, such as input-convex neural networks [AXK17, DW25, LO25], which we build on.

Non-parametric PCF fitting problems. We can solve several more general fitting problems exactly, with no constraint on the architecture (*i.e.*, non-parametric) and no regularization. When $p = 0$ and the loss is convex in y, the problem of minimizing the average loss over all convex functions, which is an infinite-dimensional non-parametric fitting problem, can be solved exactly using convex optimization, as decribed in [BV04, §5.5.5].

This can be extended to the more general case when $p > 0$ and $r = 0$. To do this we collect the data into groups associated with unique values of θ, and for each one, we fit a convex function to these data as described above. We can then interpolate these functions for values of θ not appearing in the data. This solution globally minimizes the loss, but since it does not include any penalty on complexity of f, it is very likely to perform poorly on unseen data.

1.5 Contribution

We provide a seamless path from data to a PCF that can be used for parametrized convex optimization. Our open-source implementation LPCF offers a simple user interface for fitting a PCF to variable-parameter-output triples and a variety of extensions for PCFs with special properties and uses. The resulting PCFs can be exported for immediate use in optimization frameworks like JAX [BFH+18] or CVXPY.

1.6 Outline

In §2 we propose an architecture, which is a generalization of input-convex neural networks that handles parameters, and describe our specific implementation. In §3 we describe a set of extensions that allow for modeling more specialized types of PCFs. In §4 we give some numerical results for both simple illustrative examples and some practical ones. In §5 we conclude the paper.

2 Proposed neural network architecture

2.1 Architecture

Let $\phi : \mathbf{R} \to \mathbf{R}$ be a nondecreasing convex activation function, such as rectified linear unit (ReLU) with $\phi(a) = \max\{a, 0\}$ or softplus with $\phi(a) = \log(1 + e^a)$, that we will also refer to as *logistic* due to its use in formulating logistic regression problems. We have L layers, with $z^l \in \mathbf{R}^{n_l}$ the activation of layer l, $l = 1, \ldots, L-1$. We have

$$z^0 = x, \quad z^l = \phi\left(W^l z^{l-1} + V^l x + \omega^l\right), \quad l = 1, \ldots, L-1, \quad y = W^L z^{L-1} + V^L x + \omega^L, \tag{3}$$

where ϕ is applied componentwise. Here $W^l \in \mathbf{R}^{n_l \times n_{l-1}}$ is the weight matrix associated with layer l, $V^l \in \mathbf{R}^{n_l \times n}$ is the weight matrix that feeds the input x into layer l, and $\omega_l \in \mathbf{R}^{n_l}$ is the offset for layer l. We can take $W^1 = 0$ without loss of generality since $z^0 = x$. This is a standard residual network architecture, with feedforward from the input x into each layer [HZRS16a, HZRS16b].

We take the weight matrices W^l, V^l, and offsets ω^l to be a function of the parameter θ,

$$(W^2, \ldots, W^L, V^1, \ldots, V^L, \omega^1, \ldots, \omega^L) = \psi(\theta), \tag{4}$$

where $\psi : \Theta \to \mathbf{R}^m$ describes a generic neural network architecture with output dimension

$$m = n_2 n_1 + \cdots + n_L n_{L-1} + n_1 n + \cdots + n_L n + n_1 + \cdots + n_L.$$

(We will impose one constraint on the function ψ, described below.) We denote the weight matrices and offsets as $W^l(\theta)$, $V^l(\theta)$ and $\omega^l(\theta)$ to emphasize their dependence on the parameter θ. The model weights w defining the loss in (2) are the weights defining ψ.

The proposed architecture is given by (3) and (4). It is specified by the layer dimensions $n_1, \ldots, n_{L-1}$ (the final layer width n_L is fixed to d) and the architecture for ψ. This architecture determines a function $f : \mathbf{R}^n \times \Theta \to \mathbf{R}^d$, illustrated as a block diagram in figure 1. The model weights w associated with f are the model weights appearing in the righthand side of (4).

Convexity. We impose one restriction on the weight matrices: $W^l(\theta)$ are elementwise nonnegative, for any $\theta \in \Theta$. This can be enforced by the architecture of ψ, for example by having the weight matrices $W^l(\theta)$ come directly from a nonnegative activation function such as ReLU or logistic, without any offset.

With this restriction, the function f is a PCF. We argue using recursion that each element of z^l is a convex function of x, for any fixed θ. It is evidently true for $l = 1$. Assuming that each element of z^{l-1} is a convex function of x, we observe that each entry of

$$W^l(\theta) z^{l-1} + V^l(\theta) x + \omega^l(\theta)$$

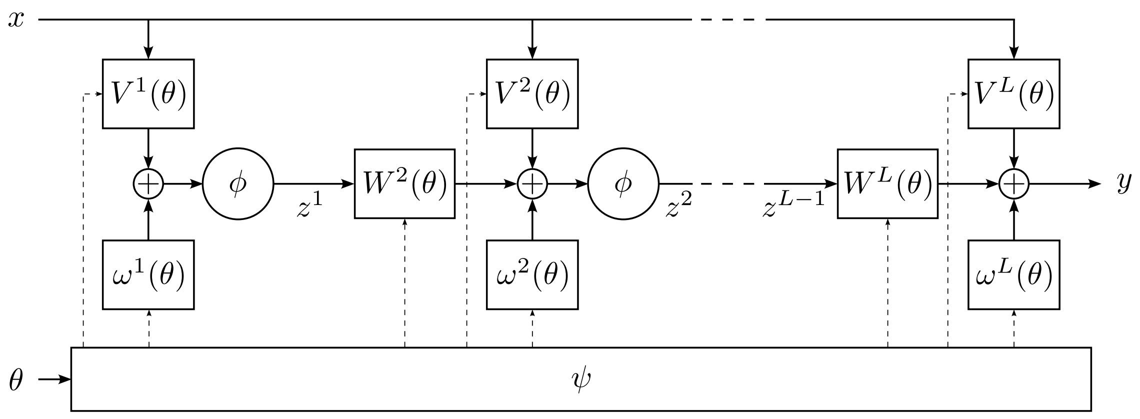

Figure 1: Neural network architecture for PCF $y = f(x, \theta)$.

is a convex function of x, since it is a nonnegative weighted sum of convex functions, plus an affine function of x. By the composition rule [BV04, §3.2.4], each entry of z^l is a convex function of x. This argument is exactly the one used in DCP. This means that, assuming the activation function is an atom, $f(x, \theta)$ is DCP.

As in an input convex neural network, our architecture (and the nonnegativity constraint on W^l), was chosen specifically so that it implements a PCF for any valid choice of the weights.

Loss and regularizer. The loss and regularizer used in the fitting process are arbitrary. Some conventional choices of loss include quadratic, ℓ_1, or Huber [Hub92, HR11]. Conventional choices of regularizer include quadratic, ℓ_1, or a combination (known as elastic net regularization). See, *e.g.*, [ZH05] or [BV04, Chap. 6.3].

2.2 Implementation

Our open-source implementation `lpcf` is available at

`https://github.com/cvxgrp/lpcf`.

In `lpcf` the user can specify the architecture, activation function, and type of regularization. The generic `.fit(data)` method fits the PCF to data, using cross-validation to choose the regularization hyper-parameter, and reports the performance of the fit obtained, for separate test data. In `lpcf`, the R2-score [Dra98, Pea05] is the standard validation metric used for cross-validation and testing. Utilities are provided for the user to evaluate the resulting PCF, score it on another data set, and to export the function to CVXPY, where it can be freely used anywhere a convex function can be.

A simple script illustrating this functionality is shown in figure 2. In lines 5–6 we instantiate a `pcf` object and fit its weights to data. In line 11 we export to the DCP expression `f` via the `tocvxpy` method, which takes a CVXPY variable and a CVXPY parameter as arguments, which are defined in the two lines above. We illustrate the use of `f` in constructing a

```python
1  from lpcf.pcf import PCF
2  import cvxpy as cp
3
4  # create default PCF object and fit to data Y, X, Theta
5  pcf = PCF()
6  pcf.fit(Y, X, Theta)
7
8  # export to cvxpy expression
9  x = cp.Variable((n, 1))
10 theta = cp.Parameter((p, 1))
11 f = pcf.tocvxpy(x, theta)
12
13 # solve cvxpy problem involving f
14 # g is another function, constraints a list of (in)equalities
15 problem = cp.Problem(cp.Minimize(f + g), constraints)
16 theta.value = ...
17 problem.solve()
18
```

Figure 2: Using LPCF with CVXPY. The initialization code for dimensions `n`, `p`, data `Y`, `X`, `Theta`, and CVXPY objects `g` and `constraints` is omitted for clarity.

CVXPY problem in line 15. In this example we simply add the fitted PCF to the objective, but we note that it can be used anywhere in CVXPY that a convex function appear, *e.g.*, in constraints.

To fit the weights w of the network ψ we use `jax-sysid` [Bem24], a Python package based on the auto-differentiation framework JAX, for system identification, neural network training, and nonlinear regression/classification. The package supports several types of regularization, simple bounds on w, parallel training from different initial values of w, and quasi-Newton methods for faster terminal convergence and better model quality than gradient descent.

Default choices. We make a number of default choices, all of which can be modified by the user. We take $L = 3$ layers in the input-convex network, with layer dimensions $n_1 = \cdots = n_{L-1} = 2\lfloor (n+d)/2 \rfloor$, and ReLU activation function. We take ψ to be a fully connected neural network with feedforward terms from θ into each layer, very similar to the architecture of the input-convex network. The output activation ϕ_W only affects $W^2, \dots, W^L$. We use the ReLU function to make them elementwise nonnegative. The architecture is visualized in figure 3. By default there are $M = 3$ layers, of which the two inner ones are $\lfloor (p+m)/2 \rfloor$ wide, with ReLU activation. By default, we use a quadratic loss (mean squared error) without regularization, *i.e.*, $\lambda = 0$, and the cross-validation procedure is turned off. Otherwise, the default number of folds for cross-validation is 5. The default optimizer runs 200 iterations

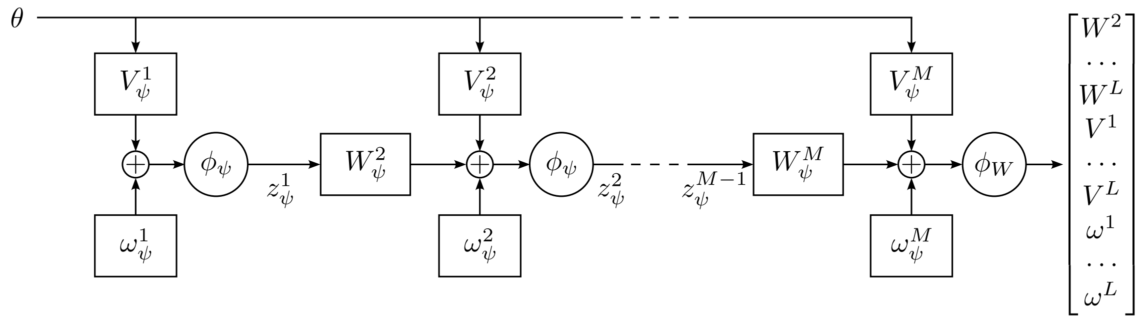

Figure 3: Neural network architecture for $\psi(\theta)$.

of Adam [Kin14] to obtain a good set of network weights, followed by 2000 iterations of L-BFGS-B [BLNZ95] (a variant of L-BFGS for bound-constrained optimization, to respect the nonnegativity constraints on W^l) to refine the model weights. We run the entire fitting method on multiple random initial sets of model weights, and take the best one as our final choice. We parallelize training with multi-core processing, with a default of 4 cores. By default, we set the number of initializations to 10 or the number of cores, whichever is greater. All of these settings can be customized by the user.

3 Extensions

We describe a few extensions of the basic model and methods.

3.1 Adding a quadratic term

We can add a (convex) quadratic function to our more general architecture (3),

$$y = x^T Q x + W^L z^{L-1} + V^L x + \omega^L,$$

where $Q \in \mathbf{S}_+^n$. We use the representation $Q = U^T U$ with upper-triangular $U \in \mathbf{R}^{n\times n}$, and emit U from $\psi(\theta)$, as in

$$(W^2, \dots, W^L, V^1, \dots, V^L, \omega^1, \dots, \omega^L, U) = \psi(\theta).$$

This automatically guarantees positive semidefiniteness of Q, without any additional constraints on the parameter network ψ.

A further variation is a low rank plus diagonal quadratic $Q = F^T F + \mathbf{diag}(d_1^2, \dots, d_n^2)$, where $F \in \mathbf{R}^{m\times n}$ is a wide matrix with $m \ll n$ and $d \in \mathbf{R}^n$ is a vector of diagonal entries before they are squared to make Q positive semidefinite. Such a form can be used to capture the dominant directions of curvature when n is large.

3.2 Monotonicity

In certain cases we may wish to impose that the PCF f is monotonically nondecreasing or nonincreasing with respect to some or all the components of x for any $\theta \in \Theta$, *e.g.*, when modeling a concave and increasing utility function (that we wish to maximize).

Monotonicity of f can be imposed as follows. Assume that the activation function ϕ is nondecreasing (*e.g.*, ReLU or softplus). Consider the architecture of the network ψ described in figure 3. We achieve monotonicity by obtaining V^j, $j = 1, \ldots, L$ similary to how we obtain W^j, as the output of the nonnegative activation ϕ_W, $j = 2, \ldots, L$ (enforcing convexity in that case).

We can prove by induction that the resulting function f is monotonically increasing with respect to each component x_i, $i = 1, \ldots, n$, for any $\theta \in \Theta$. First, $z^1 = \phi(W^1 x + V^1 x + \omega^1) = \phi(V^1 x + \omega^1)$ is increasing, since ϕ is increasing and its argument is affine and increasing with respect to each x_i; assuming that $z^l = \phi(W^l z^{l-1} + V^l x + \omega^l)$ is increasing in x_i, then $W^{l+1} z^l + V^{l+1} x + \omega^{l+1}$ remains increasing since both W^{l+1} and V^{l+1} are nonnegative. Since ϕ is increasing, it follows that z^{l+1} is increasing in x_i.

By cascading instead V^j with a nonpositive activation function ϕ_-, such as $\phi_- = -\phi_+$, $j = 1, \ldots, L$, by repeating the above argument it is easy to see that the resulting PCF f is convex and decreasing with respect to each component of x for any $\theta \in \Theta$.

3.3 Specifying a subgradient

We may wish to specify (or encourage) a PCF to be minimized at a particular point $g(\theta)$, where $g : \Theta \to \mathbf{R}^n$ is given. When f is differentiable this is equivalent to $\nabla_x f(g(\theta), \theta) = 0$, or, more generally, that $0 \in \partial_x f(g(\theta), \theta)$ (the subdifferential of f with respect to x). This can be encouraged in the training process by adding a regularization term such as

$$\ell_{\min}(w) = \frac{\rho_{\min}}{N} \sum_{i=1}^{N} \left\| \nabla_x f(g(\theta^k), \theta^k) \right\|_2^2 \tag{5}$$

to the loss (2), where $\rho_{\min}$ is a hyper-parameter that scales the regularization. This (sub)gradient can be computed by automatic differentiation of the PCF f with respect to its first argument.

As a further generalization, we can require (or encourage) $f(x, \theta) - q(\theta)^T x$ to be minimized at $x = g(\theta)$. This is equivalent to $\nabla_x f(g(\theta), \theta) = q(\theta)$, and a similar regularization term can be added to the training objective.

3.4 Fitting a parametrized convex set

The same framework can be used to fit a parametrized convex set $C : \Theta \to 2^{\mathbf{R}^n}$, described as

$$C(\theta) = \{x \mid f(x, \theta) \leq 0\},$$

where f is a PCF. Our data has the form (x^k, θ^k, y^k), $k = 1, \ldots, N$, with $y^k \in \{-1, 1\}$, where $y^k = -1$ means that $x^k \notin C(\theta^k)$ and $y^k = 1$ means that $x^k \in C(\theta^k)$. We use the logistic loss function

$$\ell(w; x^k, \theta^k, y^k) = \log(1 + e^{-y^k f(x^k, \theta^k)}).$$

We can judge the loss on test data using either the same logistic loss function or the actual error rate,

$$\ell^{\text{test}}(w; x^k, \theta^k, y^k) = \begin{cases} 0 & y^k f(x^k, \theta^k) \geq 0 \\ 1 & y^k f(x^k, \theta^k) < 0. \end{cases}$$

4 Experiments

We start with two illustrative examples on simple functions, and then give two real applications, battery aging and control. We conduct the experiments on an Apple M4 Max machine.

4.1 Piecewise affine function on R

We generate data from a piecewise affine (PWA) function of the form

$$f^{\text{true}}(x, \theta) = s_+ \max\{0, x - m\} + s_- \max\{0, m - x\} + v,$$

where $x \in \mathbf{R}$ and $\theta = (s_+, s_-, m, v) \in \mathbf{R}^4$. Note that f^{true} is convex (*i.e.*, a PCF) when $s_+ \geq -s_-$, but we also carry out experiments when this is not the case. We use a quadratic loss function.

Experimental setup. We sample 2000 values of θ from a uniform distribution on $[-1, 1]^4$. For each value of θ, we take 50 equally spaced points x on $[-1, 1]$, which gives $N = 2000 \times 50 = 10^5$ data points. We divide these data into two sets: Those for which f^{true} is convex, and those for which it is not. For those that are nonconvex, we find the best affine fit, which is the (non-parametric) closest convex approximation of the function.

We fit this data using the default values for the network architecture and learning parameters.

Results. Table 1 contains the root mean squared error (RMSE) values of the learned PCF f, for 10^5 random test data points with the true function values on a scale from -3 to 3. The first three values are with respect to the true data-generating function f^{true}, for all test data and computed for the convex and nonconvex data separately. The last value is the RMSE for nonconvex data, computed with the best affine fit as the ground truth. We observe that the overall RMSE is very small. When only considering the convex examples, the RMSE is more than an order of magnitude smaller. As expected, the RMSE is considerably larger when only taking the nonconvex examples. When computed with respect to the best affine fit f^{linear}, the RMSE for the nonconvex examples is also in the order of the RMSE for the convex data.

Figure 4 shows the true data-generating function f^{true} and the learned PCF f, for four random values of θ. For the first two parameter values θ^1 and θ^2, f^{true} is convex; for the last

Data	RMSE
All	0.054
Convex	0.001
Nonconvex	0.077
Nonconvex (relative to best affine fit)	0.004

Table 1: RMSE values. The value of f^{true} ranges between -3 and 3. The last entry compares the PCF approximation for nonconvex data to the best affine fit.

two parameter values θ^3 and θ^4, it is not convex. For the nonconvex pair we also show the best affine fit. We can see that the approximations are very good in both cases.

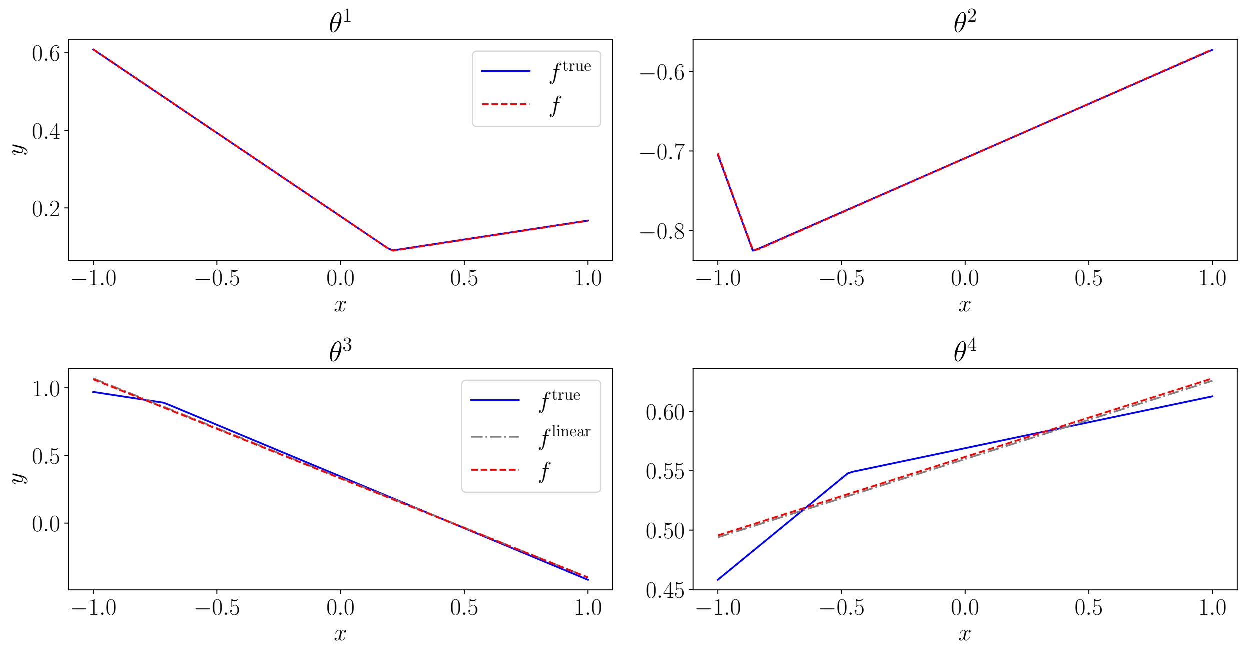

Figure 4: Data-generating function f^{true} and learned PCF f for four parameter values. *Top.* Convex f^{true}. *Bottom.* Non-convex f^{true}.

4.2 Quadratic function on $\mathbf{R}^3$

We generate data from a parametrized quadratic function

$$f^{\text{true}}(x, \theta) = x^T \theta x,$$

where the variable is $x \in \mathbf{R}^n$ and the parameter is $\theta \in \mathbf{S}^n_+$ (the set of $n \times n$ positive semidefinite symmetric matrices). The true function f^{true} is convex, *i.e.*, a PCF.

Experimental setup. We take $n = 3$ and generate 1000 values of θ where each entry is taken from $[-1, 1]$. For each value of θ, we sample 100 values of x from a uniform distribution on the unit ball, which gives $N = 10^5$ data points.

We fit the PCF with the softplus activation function and otherwise default settings for the network architecture and learning algorithm.

Results. For 10^5 random test data points, where the value of the true data generating function f^{true} moves between 0 and 2, we attain a low RMSE value of about 0.02.

4.3 Battery aging

When repeatedly charging and discharging the battery of, *e.g.*, a hybrid electric vehicle or an industrial energy storage system, the battery capacity degrades over time, a process called battery aging [SO16, SOS+11, EEG12]. We denote by y the aging rate and generate data from the battery aging model as introduced in [SO16] and used in [NOBL25] for optimal battery management,

$$f^{\text{true}}(x,\theta) = zA^{z-1}b(\alpha q/Q+\beta)\exp\left(\frac{-E_a+\eta b/Q}{R_g(T_0+T)}\right),$$

where $x=(q,b)\in\mathbf{R}^2_+$ is the variable, consisting of the battery's charge q and the absolute charge rate b. The parameter is $\theta=(A,Q,T)\in\mathbf{R}^3_+$ and contains the accumulated charge throughput A, the battery capacity Q, and temperature T. The remaining symbols are constants. We use the values in [NOBL25], the physical constants

$$E_a = 31500, \quad R_g = 8.3145, \quad T_0 = 273.15,$$

and battery parameters

$$\alpha = 28.966, \quad \beta = 74.112, \quad z = 0.6, \quad \eta = 152.5.$$

Experimental setup. We generate 1000 values of θ, with accumulated charge throughput $A\in[0,50]$ and temperature $T\in[10,50]$. We keep the battery capacity fixed at $Q=1$, since it is the slowest changing parameter (especially for a new battery where A moves fast) and one can re-fit the PCF as Q changes over time. For each value of θ, we sample 100 values of x with state of charge $q\in[0.2,0.8]$ and charge rate $b\in[0,30]$, which gives $N=10^5$ data points.

We fit the PCF with the softplus activation function. The input-convex network is 5 wide, the network ψ has a single hidden layer of width 10, and we train with 1000 and 4000 epochs with Adam and L-BFGS-B, respectively.

We compare our fit to the convex approximation used in [NOBL25] for short term battery management,

$$f^{\text{short}}(x,\theta) = \mu(1+\nu Q/2)b,$$

where

$$\mu = \beta\exp\left(\frac{-E_a}{R_g(T_0+T)}\right)zA^{z-1}, \quad \nu = \frac{\alpha}{\beta Q}.$$

This is a PCF itself, and can be derived as the first-order Taylor expansion of f^{true} around the point $(q,b)=(Q/2,0)$ [NOBL25].

Results. For 10^5 random test points with f^{true} on a scale from 0 to 0.06, the RMSE is 0.001. This is about 7 times better than using the short term approximation f^{short}, which gives an RMSE of 0.007. Figure 5 shows the true data-generating function f^{true} and the learned PCF f, for three random values of θ. We observe that the approximations are good in all cases.

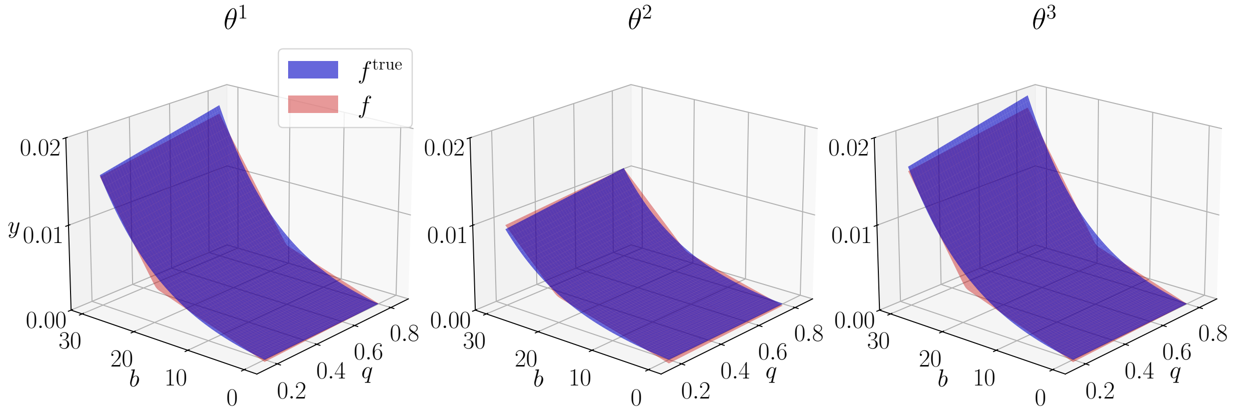

Figure 5: Data-generating function f^{true} and learned PCF f for three parameter values.

4.4 Approximate dynamic programming

We consider a nonlinear dynamical system of the form

$$z_{t+1} = F(z_t, \theta) + G(z_t, \theta)u_t, \quad t = 0, 1, \ldots, \tag{6}$$

where $z_t \in \mathbf{R}^n$ is the state, $u_t \in \mathbf{R}^m$ is the input, θ is a vector of parameters, $F : \mathbf{R}^n \times \Theta \to \mathbf{R}^n$ gives the dynamics, and $G : \mathbf{R}^n \times \Theta \to \mathbf{R}^{n\times m}$ gives the input-to-state matrix. Since z_{t+1} is an affine function of u_t, this is called input-affine form. We are given the inital state z_0, and seek inputs $u_0, u_1, \ldots$ that minimize the cost function

$$J(z_0) = \sum_{t=0}^{\infty} H(z_t, u_t, \theta), \tag{7}$$

where $H : \mathbf{R}^n \times \mathbf{R}^m \times \Theta \to \mathbf{R}_+$ is a nonnegative stage cost, assumed convex with respect to (z_t, u_t). This is a nonconvex optimal control problem, which is hard to solve globally. A local method can however be used to approximately solve this problem, typically out to some large terminal time $t = T$.

Approximate dynamic programming (ADP) is a heuristic for approximately solving the optimal control problem, choosing u_t one step at a time by solving a convex problem. It has the form

$$u_t = \operatorname*{argmin}_u \left(H(z_t, u, \theta) + \hat{V}(F(z_t, \theta) + G(z_t, \theta)u, \theta) \right), \tag{8}$$

for $t = 0, 1, \ldots$, where we update z_t using the dynamics (6). Here $\hat{V} : \mathbf{R}^n \times \Theta \to \mathbf{R}$ is a PCF, so the minimization over u is a convex optimization problem.

ADP is motivated by the Bellman or recursive form of the optimal input sequence u_t, given by (8) with $\hat{V}$ replaced by the value or cost-to-go function

$$V(z_0, \theta) = \min_{u_0, u_1, \ldots} J(z_0, \theta)$$

(see, *e.g.*, [Ber05, Ste17, AZB24]).

To find the convex approximate value function, we use a local method to approximately solve the problem (out to some large time period $t = T$) for multiple values of the initial state z_0. Each such optimization in fact gives us a set of (approximate) values of the value function, one obtained at each state on the trajectory, with value equal to the corresponding tail cost.

Experimental setup. We consider the problem of swinging up a pendulum to the vertical position by controlling the applied torque. We start with a nonlinear model of the continuous time dynamics

$$ml^2\ddot{\delta} + b\dot{\delta} + mgl \sin\delta = u,$$

where δ is the angular position of the pendulum, $m \in [0.5, 2]$ is its mass, $l = 1$ its length, $b = 0.05$ is the damping coefficient, $g = 9.81$ is the gravitational acceleration, and u is the applied torque, all in standard metric units. We obtain an input-affine discrete-time model as in (6) by setting $z = (\delta, \dot{\delta})$ and using a first-order forward Euler integration (with sampling time 0.02). The stage cost is

$$H(z, u) = (\delta - \pi)^2 + 0.01\dot{\delta}^2 + 0.001u^2.$$

Our parameter is $\theta = m$, with $\Theta = [0.5, 2]$.

We generate $N = 1000$ data points by sampling uniformly $\delta_k \in [-\pi/6, 7\pi/6]$, $\dot{\delta}_k \in [-1, 1]$, and $m_k \in [0.5, 2]$. For each combination, we (approximately) solve the optimal control problem (7) out to $t = T = 150$, using the L-BFGS-B optimizer.

We fit the PCF with the softplus activation function, width 20 for the input-convex network, two hidden layers of width 10 for the network ψ, and default architecture otherwise. We use the elastic net regularization term $r(w) = 10^{-8}\|w\|_2^2 + 0.1\|w\|_1$ in (2) and two of the extensions described in §3: We add a quadratic term (see §3.1) and specify a subgradient (see §3.3). In particular, we know that $V(z, \theta)$ has a minimum at the equilibrium $z^{\rm eq} = (\pi, 0)$. We promote that the same holds for f, by using $g(\theta) = z^{\rm eq}$ in the regularization term (5).

We train the PCF from 16 different initial values, running 1000 Adam iterations followed by up to 5000 function evaluations in L-BFGS-B.

Results. Figure 6 compares the input $\hat{u}_0$ of the ADP controller (obtained by solving the convex approximation (8), which uses the PCF f) and the input $u_0^\star$ obtained by (approximately) solving the nonlinear optimal control problem (7) out to $t = T = 150$, for training and test data (generated similarly to training data). We further compare the performance of the ADP controller to that of the nonlinear controller, for swinging up the pendulum from the initial state $z_0 = (0, 0)$, and $\theta = m = 1$. The resulting trajectories are shown on the right of figure 6. Albeit the simplicity of the ADP controller, it behaves similarly to the nonlinear controller.

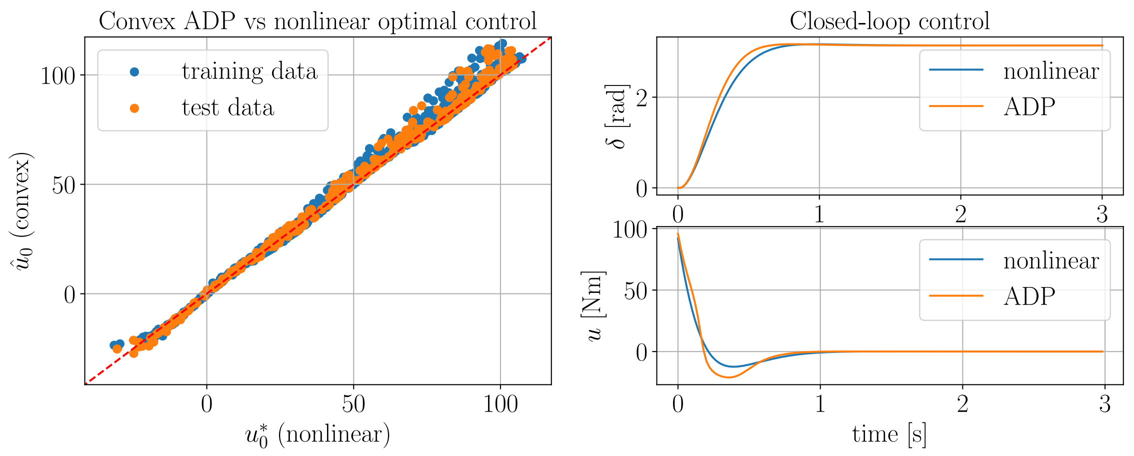

Figure 6: Left: ADP control input $\hat{u}_0$ versus nonlinear control input $u_0^\star$. Right: Simulation of the two controllers starting from $z_0 = (0, 0)$, where $\theta = m = 1$.

5 Conclusions

We have shown how to fit a PCF to data, in a simple yet customizable way, with our open-source Python tool LPCF. Our method allows (parametrized) convex optimization to be (in part) data driven: while the modeling stage may heavily rely on learning functions from data, the first-principles structure of convex optimization is retained when solving a given problem instance. Our experiments exhibit good modeling accuracy (also compared to alternative convex function approximations), and the use of learned PCFs in convex optimization problems.

References

[AVDB18] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[AXK17] B. Amos, L. Xu, and Z. Kolter. Input convex neural networks. In *International conference on machine learning*, pages 146–155. PMLR, 2017.

[AZB24] S. Abdufattokhov, M. Zanon, and A. Bemporad. Learning Lyapunov terminal costs from data for complexity reduction in nonlinear model predictive control. *Int. J. Robust Nonlinear Control*, 34(13):8676–8691, 2024.

[BAB20] S. Boyd, A. Agrawal, and S. Barratt. Embedded convex optimization for control. In *Proceedings 59th IEEE Conference on Decision and Control*. IEEE, 2020.

[Bem24] A. Bemporad. Linear and nonlinear system identification under ℓ_1- and group-Lasso regularization via L-BFGS-B. *Submitted for publication*, 2024. Also available on arXiv at `http://arxiv.org/abs/2403.03827`.

[Ber05] D. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, Belmont, MA, 3 edition, 2005.

[BFH$^+$18] J. Bradbury, R. Frostig, P. Hawkins, M. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: Composable transformations of Python+NumPy programs, 2018. Available at `http://github.com/jax-ml/jax`.

[BLNZ95] R. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[DB16] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[DHL17] I. Dunning, J. Huchette, and M. Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[Dra98] N. Draper. *Applied regression analysis*. McGraw-Hill. Inc, 1998.

[DW25] T. Deschatre and X. Warin. Input convex Kolmogorov Arnold networks. *arXiv preprint arXiv:2505.21208*, 2025.

[EEG12] S. Ebbesen, P. Elbert, and L. Guzzella. Battery state-of-health perceptive energy management for hybrid electric vehicles. *IEEE Transactions on Vehicular technology*, 61(7):2893–2900, 2012.

[FNB20] A. Fu, B. Narasimhan, and S. Boyd. CVXR: An R package for disciplined convex optimization. *Journal of Statistical Software*, 94(14):1–34, 2020.

[GB14] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.1, 2014.

[GPM89] C. Garcia, D. Prett, and M. Morari. Model predictive control: Theory and practice – a survey. *Automatica*, 25(3):335–348, 1989.

[HR11] P. Huber and E. Ronchetti. *Robust statistics.* John Wiley & Sons, 2011.

[Hub92] P. Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics: Methodology and distribution*, pages 492–518. Springer, 1992.

[HZRS16a] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778. IEEE, 2016.

[HZRS16b] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 630–645. Springer, 2016.

[KC16] B. Kouvaritakis and M. Cannon. *Model predictive control.* Springer, 2016.

[Kin14] D. Kingma. ADAN: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[LÖ4] J. Löfberg. YALMIP: A toolbox for modeling and optimization in Matlab. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 284–289. IEEE, 2004.

[LO25] Y. Liu and F. Oliveira. ICNN-enhanced 2SP: Leveraging input convex neural networks for solving two-stage stochastic programming. *arXiv preprint arXiv:2505.05261*, 2025.

[NOBL25] O. Nnorom, G. Ogut, S. Boyd, and P. Levis. Aging-aware battery control via convex optimization. *arXiv preprint arXiv:2505.09030*, 2025.

[Pea05] K. Pearson. *On the general theory of skew correlation and non-linear regression.* Dulau and Company, 1905.

[RMD$^+$17] J. Rawlings, D. Mayne, M. Diehl, et al. *Model predictive control: Theory, computation, and design*, volume 2. Nob Hill Publishing Madison, WI, 2017.

[SO16] G. Suri and S. Onori. A control-oriented cycle-life model for hybrid electric vehicle lithium-ion batteries. *Energy*, 96:644–653, 2016.

[SOS$^+$11] L. Serrao, S. Onori, A. Sciarretta, Y. Guezennec, and G. Rizzoni. Optimal energy management of hybrid electric vehicles including battery aging. In *Proceedings of the 2011 American control conference*, pages 2125–2130. IEEE, 2011.

[Ste17] B. Stellato. *Mixed-integer optimal control of fast dynamical systems.* PhD thesis, University of Oxford, 2017.

[UMZ$^+$14] M. Udell, K. Mohan, D. Zeng, J. Hong, S. Diamond, and S. Boyd. Convex optimization in Julia. *SC14 Workshop on High Performance Technical Computing in Dynamic Languages*, 2014.

[ZH05] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 67(2):301–320, 2005.