DC3: A LEARNING METHOD FOR OPTIMIZATION WITH HARD CONSTRAINTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large optimization problems with hard constraints arise in many settings, yet classical solvers are often prohibitively slow, motivating the use of deep networks as cheap "approximate solvers." Unfortunately, naive deep learning approaches typically cannot enforce the hard constraints of such problems, leading to infeasible solutions. In this work, we present Deep Constraint Completion and Correction (DC3), an algorithm to address this challenge. Specifically, this method enforces feasibility via a differentiable procedure, which implicitly completes partial solutions to satisfy equality constraints and unrolls gradient-based corrections to satisfy inequality constraints. We demonstrate the effectiveness of DC3 in both simple quadratic programming tasks and the real-world setting of AC optimal power flow, where hard constraints encode the physics of the electrical grid. In both cases, DC3 achieves near-optimal objective values while preserving feasibility.

1 INTRODUCTION

Traditional approaches to constrained optimization are often expensive to run for large problems, necessitating the use of inexpensive function approximators. Neural networks are fast to compute and have high expressivity, making them ideal as function approximators. However, while deep learning has proven its power for unconstrained problem settings, it has struggled to perform well in domains where it is necessary to satisfy hard constraints at test time. For example, in power systems, weather and climate models, materials science, and many other areas, data follows well-known physical laws, and violation of these laws can lead to answers that are unhelpful or even nonsensical. There is thus a need for fast, optimal neural network approximators that can operate in settings where traditional optimizers are slow (such as non-convex optimization), yet where strict feasibility criteria must be satisfied.

In this work, we introduce Deep Constraint Completion and Correction (DC3), a framework for applying deep learning to optimization problems with hard constraints. Our approach embeds differentiable operations into the training of the neural network to ensure feasibility. Specifically, the network outputs a partial set of variables with codimension equal to the number of equality constraints, and completes this partial set into a full solution. This completion process guarantees feasibility with respect to the equality constraints and is differentiable (either explicitly, or via the implicit function theorem). We then fix any violations of the inequality constraints via a differentiable correction procedure based on gradient descent. Together, this process of completion and correction ensures feasibility with respect to all constraints. Further, this process is fully differentiable and can be incorporated into standard deep learning methods.

Our key contributions are:

- **Framework for incorporating hard constraints.** We describe a general framework, DC3, for incorporating (potentially non-convex) equality and inequality constraints into deep-learning-based optimization algorithms.
- **Practical demonstration of feasibility.** We implement the DC3 algorithm in both convex and non-convex optimization settings. We demonstrate the success of the algorithm in producing approximate solutions with significantly greater feasibility than other deep learning approaches, while maintaining near-optimality of the solution.

- **AC optimal power flow.** We show how the general DC3 framework can be used to optimize power flows on the electrical grid. This difficult non-convex optimization task must be solved at scale and is especially critical for renewable energy adoption. Our results greatly improve upon the performance of general-purpose deep learning methods on this task.

2 Related Work

Our approach is situated within the broader literature on fast optimization methods, and draws inspiration from literature on implicit layers and on incorporating constraints into neural networks. We briefly describe each of these areas and their relationship to the present work.

Fast optimization methods. Many classical optimization methods have been proposed to improve the practical efficiency of solving optimization problems. These include general techniques such as constraint and variable elimination (i.e., the removal of non-active constraints or redundant variables, respectively), as well as problem-specific techniques (e.g., KKT factorization techniques in the case of convex quadratic programs) (Nocedal & Wright, 2006). Our present work builds upon aspects of this literature, applying concepts from variable elimination to reduce the number of degrees of freedom associated with the optimization problems we wish to solve.

In addition to the classical optimization literature, there has been a large body of literature in deep learning that has sought to approximate or speed up optimization models. As described in reviews on topics such as combinatorial optimization (Bengio et al., 2020) and optimal power flow (Hasan et al., 2020), machine learning methods to speed up optimization models have thus far taken two main approaches. The first class of approaches, akin to work on surrogate modeling (Koziel & Leifsson, 2013), has involved training machine learning models to map directly from optimization inputs to full solutions. However, such approaches have often struggled to produce solutions that are both feasible and (near-)optimal. The second class of approaches has instead focused on employing machine learning approaches alongside or in the loop of optimization models, e.g., to learn warm-start points (e.g., see Baker (2019)) or to enable constraint elimination techniques by predicting active constraints (e.g., see Misra et al. (2018)). We view our work being part of the former set of approaches, but drawing important inspiration from the latter: that employing structural knowledge about the optimization model is paramount to achieving both feasibility and optimality.

Constraints in neural networks. While deep learning is often thought of as wholly unconstrained, in reality, it is quite common to incorporate (simple) constraints within deep learning procedures. For instance, softmax layers encode simplex constraints, sigmoids instantiate upper and lower bounds, ReLUs encode projections onto the positive orthant, and convolutional layers enforce translational equivariance (an idea taken further in general group-equivariant networks (Cohen & Welling, 2016)). Recent work has also focused on embedding specialized kinds of constraints into neural networks, such as conservation of energy (see, e.g., Greydanus et al. (2019) and Beucler et al. (2019)), and homogeneous linear inequality constraints (Frerix et al., 2020). However, while these represent common "special cases," there has to date been little work on building more general hard constraints into deep learning models.

Implicit layers. In recent years, there has been a great deal of interest in creating structured neural network layers that define implicit relationships between their inputs and outputs. For instance, such layers have been created for SAT solving (Wang et al., 2019), ordinary differential equations (Chen et al., 2018), normal and extensive-form games (Ling et al., 2018), rigid-body physics (de Avila Belbute-Peres et al., 2018), sequence modeling (Bai et al., 2019), and various classes of optimization problems (Amos & Kolter, 2017; Donti et al., 2017; Djolonga & Krause, 2017; Tschiatschek et al., 2018; Wilder et al., 2018; Gould et al., 2019). (Interestingly, softmax, sigmoid, and ReLU layers can also be viewed as implicit layers (Amos, 2019), though in practice it is more efficient to use their explicit form.) In principle, such approaches could be used to directly enforce constraints within neural network settings, e.g., by projecting neural network outputs onto a constraint set using quadratic programming layers (Amos & Kolter, 2017) in the case of linear constraints, or convex optimization layers (Agrawal et al., 2019) in the case of general convex constraints. However, given the computational expense of such optimization layers, these projection-based approaches are likely to be inefficient. Instead, our approach leverages insights from this line of work by using implicit differentiation to backpropagate through the "completion" of equality constraints in cases where these constraints cannot be solved explicitly (such as in AC optimal power flow).

3 DC3: DEEP CONSTRAINT COMPLETION AND CORRECTION

In this work, we consider solving families of optimization problems for which the objectives and/or constraints vary across instances. Formally, let $x \in \mathbb{R}^d$ denote the problem data, and $y \in \mathbb{R}^n$ denote the solution of the corresponding optimization problem (where y depends on x). For any given x, our aim is then to find y solving:

$$\begin{aligned} \underset{y \in \mathbb{R}^n}{\text{minimize}} \quad & f_x(y) \\ \text{subject to} \quad & g_x(y) \leq 0 \\ & h_x(y) = 0 \end{aligned} \tag{1}$$

(where f, g, and h are potentially nonlinear and non-convex). Solving such a family of optimization problems can be framed as a learning problem, where an algorithm must predict an optimal y from the problem data x. We consider deep learning approaches to this task – that is, training a neural network N_θ, parameterized by θ, to approximate y given x.

A naive deep learning approach to approximating such a problem involves viewing the constraints as a form of regularization. That is, for training examples $x^{(i)}$, the algorithm learns to minimize a composite loss containing both the objective and two "soft loss" terms representing violations of the equality and inequality constraints:

$$\ell_{\text{soft}}(\hat{y}) = f_x(\hat{y}) + \lambda_g \| \operatorname{ReLU}(g_x(\hat{y}))\|_2^2 + \lambda_h \|h_x(\hat{y})\|_2^2. \tag{2}$$

An alternative framework (see, e.g., Zamzam & Baker (2019)) is to use supervised learning on examples $(x^{(i)}, y^{(i)})$ for which an optimum $y^{(i)}$ is known. In this case, the loss is simply $||\hat{y} - y^{(i)}||_2^2$. However, both these procedures for training a neural network can lead in practice to highly infeasible outputs (as we demonstrate in our experiments), because they do not strictly enforce constraints.

To address this challenge, we introduce the method of Deep Constraint Completion and Correction (DC3), which allows hard constraints to be integrated into the training of neural networks. In particular, the DC3 method includes the following two innovations:

Equality completion. We provide a mechanism to enforce equality constraints during training and testing, inspired by the literature on variable elimination. Specifically, rather than outputting the full-dimensional optimization solution directly, we first output a subset of the variables, and then infer the remaining variables via the equality constraints – either explicitly, or by solving an implicit set of equations (through which we can then backpropagate via the implicit function theorem).

Inequality correction. We correct for violation of the inequality constraints by mapping infeasible points to feasible points using an internal gradient descent procedure that is itself differentiable during training. This allows us to fix inequality violations while taking steps along the manifold of points satisfying the equalities, which yields an output that is feasible with respect to all constraints.

Overall, our algorithm is to train a neural network $N_\theta(x)$ to output a partial set of variables z. These variables are then completed to a full set of variables $\tilde{y}$ satisfying the equality constraints. In turn, $\tilde{y}$ is corrected to $\hat{y}$ to satisfy the inequality constraints while continuing to satisfy the equality constraints. The overall network is trained using backpropagation on the soft loss described in Equation (2) (which is necessary for correction, as noted below). Importantly, both the completion and correction procedures are differentiable either implicitly or explicitly (allowing network training to take them into account), and the overall framework is agnostic to the choice of neural network architecture. Pseudocode for this full procedure is given in Algorithm 1.

We note that as this procedure is somewhat general, in cases where constraints have additional structure, specialized techniques may be more appropriate to use. For instance, while we examine linearly-constrained settings in our experiments for the purposes of illustration, in practice, techniques such as Minkowski-Weyl decomposition or Cholesky factorization (see Frerix et al. (2020), Amos & Kolter (2017)) may be more efficient in these settings. However, for more general settings without this kind of structure – e.g., for non-convex problems such as AC optimal power flow, which we examine in our experiments – the DC3 framework can provide a (differentiable) mechanism for satisfying hard constraints in practice.

We now describe the completion and correction procedures in more detail.

Algorithm 1 Deep Constraint Completion and Correction (DC3)

assume equality completion procedure $\varphi_x : \mathbb{R}^m \to \mathbb{R}^{n-m}$ *// to solve equality constraints*

procedure TRAIN(x)
  init neural network $N_\theta : \mathbb{R}^d \to \mathbb{R}^m$
  while not converged **do**
    compute partial set of variables $z = N_\theta(x)$
    complete to full set of variables $\tilde{y} = \begin{bmatrix} z^T & \varphi_x(z)^T \end{bmatrix}^T \in \mathbb{R}^n$
    correct to feasible (or approx. feasible) solution $\hat{y} = \rho_x^{(t_{\text{train}})}(\tilde{y})$
    compute constraint-regularized loss $\ell_{\text{soft}}(\hat{y})$
    update θ using $\nabla_\theta \ell_{\text{soft}}(\hat{y})$ *// e.g., via GD*
  end while
end procedure

procedure TEST(x, N_θ)
  compute partial set of variables $z = N_\theta(x)$
  complete to full set of variables $\tilde{y} = \begin{bmatrix} z^T & \varphi_x(z)^T \end{bmatrix}^T$
  correct to feasible solution $\hat{y} = \rho^{(t_{\text{test}})}(\tilde{y})$
  return $\hat{y}$
end procedure

3.1 EQUALITY COMPLETION

Assuming that the problem (1) is not overdetermined, the number of equality constraints $h_x(y) = 0$ must not exceed the dimension of the decision variable $y \in \mathbb{R}^n$: that is, the number of equality constraints equals $n - m$ for some $m \geq 0$. Then, given m of the entries of a feasible point y, the other $(n-m)$ entries are, in general, determined by the $(n-m)$ equality constraints. We exploit this observation in our algorithm, noting that it is considerably more efficient to output a point in $\mathbb{R}^m$ and complete it to a point $y \in \mathbb{R}^n$ such that $h_x(y) = 0$, as compared with outputting full-dimensional points $y \in \mathbb{R}^n$ and attempting to adjust all coordinates to satisfy the equality constraints.

In particular, we assume that given m entries of y, we either can solve for the remaining entries explicitly (e.g. in a linear system) or that we have access to a process (e.g. Newton's Method) allowing us to solve any implicit equations. Formally, we assume access to a function $\varphi_x : \mathbb{R}^m \to \mathbb{R}^{n-m}$, such that $h_x\left(\begin{bmatrix} z^T & \varphi_x(z)^T \end{bmatrix}^T\right) = 0$, where $\begin{bmatrix} z^T & \varphi_x(z)^T \end{bmatrix}^T$ is the concatenation of z and $\varphi_x(z)$.

In our algorithm, we then train our neural network N_θ to output points $z \in \mathbb{R}^m$, which are completed to $\begin{bmatrix} z^T & \varphi_x(z)^T \end{bmatrix}^T \in \mathbb{R}^n$. A challenge then arises as to how to backpropagate the loss during the training of N_θ if $\varphi_x(z)$ is not a readily differentiable explicit function – for example, if the completion procedure uses Newton's Method. We solve this challenge by leveraging the implicit function theorem, as e.g. in OptNet (Amos & Kolter, 2017) and SATNet (Wang et al., 2019). This approach allows us to express, for any training loss ℓ, the derivatives $\mathrm{d}\ell/\mathrm{d}z$ using $\mathrm{d}\ell/\mathrm{d}\varphi_x(z)$.

Namely, let $J^h \in \mathbb{R}^{(n-m)\times n}$ denote the Jacobian of $h_x(y)$ with respect to y. By the chain rule:

$$0 \;=\; \frac{\mathrm{d}}{\mathrm{d}z} h_x\left(\begin{bmatrix} z \\ \varphi_x(z) \end{bmatrix}\right) \;=\; \frac{\partial h_x(z)}{\partial z} + \frac{\partial h_x(z)}{\partial \varphi_x(z)} \frac{\partial \varphi_x(z)}{\partial z} \;=\; J^h_{:,0:m} + J^h_{:,m:n} \frac{\partial \varphi_x(z)}{\partial z},$$

which yields

$$\frac{\partial \varphi_x(z)}{\partial z} = -\left(J^h_{:,m:n}\right)^{-1} J^h_{:,0:m}. \tag{3}$$

We can then backpropagate losses through the network by noting that

$$\frac{\mathrm{d}\ell}{\mathrm{d}z} = \frac{\partial \ell}{\partial z} + \frac{\partial \ell}{\partial \varphi_x(z)} \frac{\partial \varphi_x(z)}{\partial z} = \frac{\partial \ell}{\partial z} - \frac{\partial \ell}{\partial \varphi_x(z)} \left(J^h_{:,m:n}\right)^{-1} J^h_{:,0:m}. \tag{4}$$

We note that in practice, the Jacobian $\partial\varphi_x(z)/\partial z$ should generally not be computed explicitly due to space complexity considerations; instead, it is often desirable to form the result of the left matrix-

vector product $(\partial \ell / \partial \varphi_x(z))(\partial \varphi_x(z) / \partial z)$ directly. This process is well-described in e.g. Amos & Kolter (2017), and also in detail for the problem of AC optimal power flow in Appendix C.

3.2 INEQUALITY CORRECTION

While the completion procedure described above guarantees feasibility with respect to the equality constraints, it does not ensure that the inequality constraints will be satisfied. To additionally ensure feasibility with respect to the inequality constraints, our algorithm incorporates a correction procedure that maps the outputs from the previous step into the feasible region. In particular, we employ a gradient-based correction procedure that takes gradient steps in z towards the feasible region *along the manifold of points satisfying the equality constraints*.

Let $\rho_x(y)$ be the operation that inputs a point $y = \begin{bmatrix} z^T & \varphi_x(z)^T \end{bmatrix}^T$ and moves it closer to satisfying the inequality constraints by taking a step along the gradient of the inequality violation with respect to the partial variables z. Formally, for a learning rate $\gamma > 0$, we define:

$$\rho_x\left(\begin{bmatrix} z \\ \varphi_x(z) \end{bmatrix}\right) = \begin{bmatrix} z - \gamma \Delta z \\ \varphi_x(z) - \gamma \Delta \varphi_x(z) \end{bmatrix},$$

$$\text{for} \quad \Delta z = \nabla_z \left\| \mathrm{ReLU}\left(g_x\left(\begin{bmatrix} z \\ \varphi_x(z) \end{bmatrix}\right)\right)\right\|_2^2, \quad \Delta \varphi_x(z) = \frac{\partial \varphi_x(z)}{\partial z} \Delta z.$$

While gradient descent methods do not always converge to global (or local) optima for general optimization problems, if initialized *close to an optimum*, gradient descent is highly effective in practice for non-pathological settings (see e.g. Busseti et al. (2019); Lee et al. (2017)). At test time, the input to the DC3 correction procedure should already be close to feasible with respect to the inequality constraints, as it is the output of a differentiable completion process that is trained using the soft loss ℓ_{soft}. Therefore, we may expect that in practice, the limit $\lim_{t\to\infty} \rho_x^{(t)}(y)$ will converge to a point satisfying both inequality and equality constraints (while for problems with linear constraints as in §4.1, the correction process is mathematically guaranteed to converge).

As the limit $\lim_{t\to\infty} \rho_x^{(t)}(y)$ is difficult to calculate in practice, we make approximations at both training and test time. Namely, we apply $\rho_x^{(t)}(y)$ to the output of the completion procedure, with $t = t_{\text{train}}$ relatively small at train time (to allow backpropagation through the correction) and $t = t_{\text{test}}$ relatively large at test time (to ensure convergence to a feasible point).

4 EXPERIMENTS

We evaluate DC3 both in a simple quadratic programming (QP) setting and on the real-world task of AC optimal power flow (ACOPF). In particular, we assess our method on the following criteria:

- **Optimality**: How good is the objective value $f_x(y)$ achieved by the final solution?
- **Feasibility**: How much, if at all, does the solution violate the constraints? Specifically, what are the maximum and mean feasibility violations of the inequality and equality constraints: $\max(\mathrm{ReLU}(g_x(y)))$, $\mathrm{mean}(\mathrm{ReLU}(g_x(y)))$, $\max(h_x(y))$, and $\mathrm{mean}(h_x(y))$?
- **Speed**: How fast is the method?

We compare DC3 against the following methods (referred to by abbreviations in our tables below):

- **Optimizer**: A traditional optimization solver. For QP settings, we use `OSQP` (Stellato et al., 2020), as well as the batched, differentiable solver `qpth` developed as part of OptNet (Amos & Kolter, 2017). For ACOPF, we use the solver provided by `PYPOWER`, a Python port of `MATPOWER` (Zimmerman et al., 1997).
- **NN**: A simple deep learning approach trained to minimize a soft loss (Equation (2)).

	Obj. value	Max eq.	Mean eq.	Max ineq.	Mean ineq.	Time
Optimizer (OSQP)	-15.05 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	41.68 (4.80)†
Optimizer (qpth)	-15.05 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	4.42 (1.32)
DC3	-13.85 (0.02)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.119 (0.061)
DC3, $\neq$	-12.55 (0.02)	0.35 (0.00)	0.13 (0.00)	0.00 (0.00)	0.00 (0.00)	0.034 (0.000)
DC3, $\not\leq$ train	-1.15 (1.12)	0.00 (0.00)	0.00 (0.00)	0.01 (0.01)	0.00 (0.00)	0.088 (0.000)
DC3, $\not\leq$ train/test	-0.65 (1.86)	0.00 (0.00)	0.00 (0.00)	0.22 (0.27)	0.02 (0.02)	0.001 (0.000)
DC3, no soft loss	-20.40 (0.00)	0.00 (0.00)	0.00 (0.00)	14.15 (0.06)	3.21 (0.01)	0.088 (0.001)
NN	-12.56 (0.03)	0.35 (0.00)	0.13 (0.00)	0.00 (0.00)	0.00 (0.00)	0.001 (0.000)
NN, $\leq$ test	-12.56 (0.03)	0.35 (0.00)	0.13 (0.00)	0.00 (0.00)	0.00 (0.00)	0.035 (0.000)
Eq. NN	-9.51 (0.30)	0.00 (0.00)	0.00 (0.00)	8.90 (0.88)	0.92 (0.10)	0.001 (0.000)
Eq. NN, $\leq$ test	-14.90 (0.01)	0.00 (0.00)	0.00 (0.00)	0.59 (0.03)	0.05 (0.00)	0.090 (0.001)

Table 1: Results on QP task for 100 variables, 50 equality constraints, and 50 inequality constraints. We compare the performance of DC3 and other algorithms according to the objective value and max/mean values of equality/inequality constraint violations, each averaged across test instances. We also compare the total time required to run on all 833 test instances, where all deep learning methods and `qpth` are run with full parallelization, and where we use † to denote that the baseline optimizer is run sequentially. (Std. deviations across 5 runs are shown in parentheses for all figures reported.) Lower values are better for all metrics. We find that methods other than DC3 and Optimizer violate feasibility (as shown in red). DC3 gives a feasible output with reasonable objective value about $40\times$ faster than `qpth`, and about $360\times$ faster than the `osqp` optimizer (though in the latter case, this speedup is in part because we have taken advantage of the fact that DC3 is readily parallelizable, whereas the same is not true for `osqp`).

- **NN, $\leq$ test**: The NN approach, with a gradient-based correction procedure[1] applied to the output at test time in an effort to mitigate violations of equality and inequality constraints. Unlike in DC3, correction is not used during training, and completion is not used at all.
- **Eq. NN**: A more sophisticated approach inspired by[2] that in Zamzam & Baker (2019), where (i) the neural network outputs a partial set of variables $\hat{z}$, which is completed to the full set using the equality constraints, (ii) training is performed by supervised learning on optimal pairs $(x^{(i)}, z^{(i)})$ with loss function $||\hat{z} - z^{(i)}||_2^2$, not using the objective value at all.
- **Eq. NN, $\leq$ test**: The approach in Eq. NN, augmented with gradient-based correction at test time to mitigate violations of equality and inequality constraints.

In addition, we consider weaker versions of DC3 in which components of the algorithm are ablated:

- **DC3, $\neq$**. The DC3 algorithm with completion ablated. All variables are output by the network directly and correction is performed by taking gradient steps for both equality and inequality constraints.
- **DC3, $\not\leq$ train**. The DC3 algorithm with correction ablated at train time. Correction is still performed at test time.
- **DC3, $\not\leq$ train/test**. The DC3 algorithm with correction ablated at both train and test time.
- **DC3, no soft loss**. The DC3 algorithm with training performed to minimize the objective value, without auxiliary terms capturing equality and inequality violation.

As our overall framework is agnostic to the choice of neural network architecture, to facilitate comparison, we use a fixed neural network architecture across all experiments: fully connected with two

[1]Note that this correction procedure is not exactly the same as that described in Section 3.2, as the outputs of the NN baseline do not necessarily meet the prerequisite of satisfying the equality constraints. Instead, we adjust the full set of output variables directly with respect to gradients of the inequality and equality violations.

[2]In Zamzam & Baker (2019), the authors employ one step of an ACOPF-specific heuristic called PV/PQ switching to correct inequality constraint violations at test time. We do not apply this heuristic here in the spirit of presenting a more general framework. As PV/PQ switching is not necessarily guaranteed to correct all inequality violations (although it can work well in practice), in principle, one could consider employing a combination of PV/PQ switching *and* gradient-based corrections in the context of ACOPF.

hidden layers of size 200, including ReLU activation, batch normalization, and dropout (with rate 0.2) at each hidden layer (Ioffe & Szegedy, 2015; Srivastava et al., 2014). For our correction procedure, we use $t_{\text{train}} = 100$ for the QP tasks and $t_{\text{train}} = 5$ for ACOPF, and $t_{\text{test}} = 100$ in both cases (see Appendix B). All neural networks are trained using PyTorch (Paszke et al., 2019).

To generate timing results, all neural nets and the `qpth` optimizer were run with full parallelization on a GeForce GTX 1080 Ti GPU, and the `osqp` and `PYPOWER` optimizers were run sequentially on an Intel Xeon 2.60GHz CPU. For these and many additional reasons (e.g., our DC3 implementation is not tightly optimized), we emphasize that all timing comparisons are approximate.

4.1 QUADRATIC PROGRAMS

As a first test of the DC3 method, we consider solving convex quadratic programs with a quadratic objective function and linear constraints. Formally, the problem we consider is given by:

$$\underset{y \in \mathbb{R}^n}{\text{minimize}} \ \ \frac{1}{2} y^T Q y + p^T y, \ \ \text{s.t.} \ \ Ay = x, \ Gy \leq h, \tag{5}$$

for constants $Q \in \mathbb{R}^{n \times n} \succeq 0, p \in \mathbb{R}^n, A \in \mathbb{R}^{n_{\text{eq}} \times n}, G \in \mathbb{R}^{n_{\text{ineq}} \times n}, h \in \mathbb{R}^{n_{\text{ineq}}}$, and variable $x \in \mathbb{R}^{n_{\text{eq}}}$ which varies between problem instances. Our algorithm must learn to approximate the optimal y given x as input.

In our experiments, we take Q to be a diagonal matrix with all diagonal entries drawn i.i.d. from the uniform distribution on $[0, 1]$, ensuring that Q is positive semi-definite. We take matrices A, G with entries drawn i.i.d. from the unit normal distribution. We assume that in each problem instance, all entries of x are in the interval $[-1, 1]$. In order to ensure that the problem is feasible, we take $h = \sum_j |(GA^+)_{ij}|$, where A^+ is the Moore-Penrose pseudoinverse of A; namely, for this choice of h, the point $y = A^+ x$ is feasible (but not, in general, optimal), because:

$$AA^+ x = x, \qquad GA^+ x \leq \sum_j \left|(GA^+)_{ij}\right| \ \text{ since } |x_j| \leq 1.$$

During training, we use examples x with entries drawn i.i.d. from the uniform distribution on $[-1, 1]$.

While we examine this quadratic programming setting for the purposes of illustration, we note that the general DC3 method may indeed not be the most efficient deep learning-based method for constraint enforcement in this setting. Specifically, as previously-discussed, more specialized techniques are available for linearly-constrained settings such as this one. As such, our goal in examining this set of quadratic programming problems is to analyze the performance of DC3 in a well-understood setting, before moving on to the more challenging (non-convex) setting of ACOPF.

We assess our method for different problem classes characterized by the above QP, letting $n = 100$ and performing experiments with different numbers of inequality and equality constraints. In particular, we evaluate settings with $n_{\text{ineq}} = 50$ and where the number of equality constraints varies as $n_{\text{eq}} = 10, 30, 50, 70, 90$, and similar settings where we fix $n_{\text{eq}} = 50$ and vary the number of inequality constraints as $n_{\text{ineq}} = 10, 30, 50, 70, 90$. We run each experiment 5 times over 10,000 examples x (with a train/test/validation ratio of 10:1:1). Hyperparameters were tuned to maximize performance for each method individually (see Appendix B). Optimality, feasibility, and timing results for the setting with $n_{\text{ineq}} = 50, n_{\text{eq}} = 50$ are reported in Table 1, and results for the remaining experiments are reported in Table 2 and (in the Appendix) Table A.1.

In our experiments, we find that DC3 preserves feasibility with respect to both equality and inequality constraints, while achieving reasonable objective values. For every baseline deep learning algorithm, on the other hand, feasibility is violated significantly for either equality or inequality constraints. As expected, "DC3 $\neq$" (completion ablated) results in violated equality constraints, while "DC3 $\not\leq$" (correction ablated) violates inequality constraints. Ablating the soft loss also results in violated inequality constraints, leading to an objective value significantly lower than would be possible were the constraints satisfied.

In terms of timing, DC3 runs about 40× faster than `qpth`, and about 350× faster than the `OSQP` optimizer. We note that this latter comparison is not apples-to-apples, as DC3 is run with full parallelization on a GPU, while `OSQP` is run sequentially on a CPU. However, we report results in this manner given that in practice, standard solvers such as `OSQP` are *not* readily fully parallelizable,

		10	30	50	70	90
Optimizers (OSQP, qpth)	Obj. val.	-27.26 (0.00)	-23.13 (0.00)	-15.05 (0.00)	-14.80 (0.00)	-4.79 (0.00)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
DC3	Obj. val.	-25.82 (0.04)	-20.74 (0.25)	-13.85 (0.02)	-13.76 (0.03)	-4.76 (0.01)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
DC3, $\neq$	Obj. val.	-22.83 (0.15)	-20.43 (0.06)	-12.55 (0.02)	-13.35 (0.03)	-5.29 (0.01)
	Max eq.	0.11 (0.01)	0.24 (0.00)	0.35 (0.00)	0.47 (0.00)	0.59 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
DC3, $\not\leq$ train	Obj. val.	-25.73 (0.05)	-20.18 (0.16)	-1.15 (1.12)	-13.72 (0.02)	-4.76 (0.00)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.01 (0.01)	0.00 (0.00)	0.00 (0.00)
DC3, $\not\leq$ train/test	Obj. val.	-25.73 (0.05)	-20.18 (0.16)	-0.65 (1.86)	-13.72 (0.02)	-4.76 (0.00)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.22 (0.27)	0.00 (0.00)	0.00 (0.00)
DC3, no soft loss	Obj. val.	-66.79 (0.00)	-40.66 (0.01)	-20.40 (0.00)	-15.58 (0.00)	-4.76 (0.01)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	122.38 (0.09)	50.46 (0.11)	14.15 (0.06)	5.70 (0.06)	0.00 (0.00)
NN	Obj. val.	-22.65 (0.13)	-20.42 (0.04)	-12.56 (0.03)	-13.34 (0.02)	-5.28 (0.01)
	Max eq.	0.11 (0.00)	0.24 (0.00)	0.35 (0.00)	0.47 (0.00)	0.59 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
NN, $\leq$ test	Obj. val.	-22.65 (0.13)	-20.42 (0.04)	-12.56 (0.03)	-13.34 (0.02)	-5.28 (0.01)
	Max eq.	0.11 (0.00)	0.24 (0.00)	0.35 (0.00)	0.47 (0.00)	0.59 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
Eq. NN	Obj. val.	-27.20 (0.02)	-22.99 (0.17)	-9.51 (0.30)	-14.63 (0.01)	-4.75 (0.01)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.46 (0.03)	1.44 (0.13)	8.90 (0.88)	0.82 (0.05)	0.00 (0.00)
Eq. NN, $\leq$ test	Obj. val.	-27.21 (0.02)	-22.99 (0.07)	-14.90 (0.01)	-14.66 (0.01)	-4.75 (0.01)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.37 (0.01)	0.66 (0.04)	0.59 (0.03)	0.72 (0.03)	0.00 (0.00)

Table 2: Results on QP task for 100 variables and 50 inequality constraints, as the number of equality constraints varies as $10, 30, 50, 70, 90$. We compare the performance of DC3 and other algorithms according to objective value and maximum equality/inequality constraint violations averaged across test instances. (Standard deviations across 5 runs are shown in parentheses.) We find that neural network baselines give significantly inferior performance (shown in red) across all problems.

whereas deep learning-based (approximate) optimizers can easily exploit massively parallel batch processing within deep learning frameworks.

4.2 AC OPTIMAL POWER FLOW

We now show how DC3 can be applied to the problem of AC optimal power flow (ACOPF). ACOPF is a fundamental problem for the operation of the electrical grid, and is used to determine how much power must be produced by each generator on the grid in order to meet demand. As the amount of renewable energy on the power grid grows, this problem must be solved more and more frequently to account for the variability of these renewable sources, and at larger scale to account for an increasing number of distributed devices (Rolnick et al., 2019). However, ACOPF is a non-convex optimization problem and classical optimizers scale poorly on it. While many specialized approaches to this problem have started to emerge, including approaches from machine learning (see, e.g., Zamzam & Baker (2019) for a discussion), we here assess the ability of our more general framework to address this problem.

Formally, a power network may be considered as a graph on b nodes, representing different locations (*buses*) within the electrical grid, and with edges weighted by complex numbers $w_{ij} \in \mathbb{C}$ (*admittances*) that represent how easily current can flow between the corresponding locations in the grid. Let $W \in \mathbb{C}^{n \times x}$ denote the graph Laplacian (or *nodal admittance matrix*). Then, the problem of ACOPF can be defined as follows: Given input variables $p_d \in \mathbb{R}^b$, $q_d \in \mathbb{R}^b$ (representing *real power* and *reactive power demand* at the various nodes of the graph), output the variables $p_g \in \mathbb{R}^b$, $q_g \in \mathbb{R}^b$

	Obj. value	Max eq.	Mean eq.	Max ineq.	Mean ineq.	Time
Optimizer	3.81 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	103.35 (4.48)†
DC3	3.82 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	2.209 (0.003)
DC3, $\neq$	3.67 (0.01)	0.08 (0.00)	0.02 (0.00)	0.00 (0.00)	0.00 (0.00)	0.852 (0.001)
DC3, $\not\leq$ train	3.82 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	2.203 (0.002)
DC3, $\not\leq$ train/test	3.82 (0.00)	0.00 (0.00)	0.00 (0.00)	0.01 (0.00)	0.00 (0.00)	0.081 (0.000)
DC3, no soft loss	3.07 (0.05)	3.04 (0.38)	0.08 (0.01)	0.09 (0.02)	0.01 (0.00)	2.252 (0.095)
NN	3.69 (0.02)	0.18 (0.00)	0.03 (0.00)	0.00 (0.00)	0.00 (0.00)	0.001 (0.000)
NN, $\leq$ test	3.69 (0.02)	0.10 (0.00)	0.02 (0.00)	0.00 (0.00)	0.00 (0.00)	0.855 (0.001)
Eq. NN	3.81 (0.00)	0.00 (0.00)	0.00 (0.00)	0.15 (0.00)	0.00 (0.00)	0.082 (0.000)
Eq. NN, $\leq$ test	3.81 (0.00)	0.00 (0.00)	0.00 (0.00)	0.15 (0.00)	0.00 (0.00)	0.938 (0.001)

Table 3: Results on ACOPF over 100 test instances. We compare the performance of DC3 and other algorithms according to the metrics described in Table 1. We find again that baseline methods violate feasibility (as shown in red), while DC3 gives a feasible and near-optimal output about $45\times$ faster than the PYPOWER Optimizer (once again, this speedup is in part because we have taken advantage of the fact that DC3 is readily parallelizable, whereas the same is not true for PYPOWER).

(representing real power and reactive power *generation*) and $v \in \mathbb{C}^b$ (representing *real* and *imaginary voltage*), according to the following optimization problem:

$$\begin{aligned} \underset{p_g \in \mathbb{R}^b,\ q_g \in \mathbb{R}^b,\ v \in \mathbb{C}^b}{\text{minimize}} \quad & p_g^T A p_g + b^T p_g \\ \text{subject to} \quad & p_g^{\min} \leq p_g \leq p_g^{\max} \\ & q_g^{\min} \leq q_g \leq q_g^{\max} \\ & v^{\min} \leq |v| \leq v^{\max} \\ & (p_g - p_d) + (q_g - q_d)i = \text{diag}(v)\overline{W}\overline{v}. \end{aligned} \tag{6}$$

More details about how we apply DC3 to the problem of ACOPF are given in Appendix C.

We assess our method on a 57-node power system test case available via the MATPOWER package. We conduct 5 runs over 1,200 input datapoints (with a train/test/validation ratio of 10:1:1). As with the QP task, hyperparameters for ACOPF were tuned to maximize performance for each method individually (see Appendix B). Optimality, feasibility, and timing results are reported in Table 3.

We find that DC3 achieves comparable objective values to the optimizer, and preserves feasibility with respect to both equality and inequality constraints. As with the QP task, for every baseline deep learning algorithm, feasibility is violated significantly for either equality or inequality constraints. Ablations of DC3 also suffer from constraint violations, though the effect is less pronounced than for the QP task, especially for ablation of the correction. We also see that DC3 runs about $45\times$ faster than the optimizer (though again, this is in part due to its ability to readily exploit parallelism within deep learning frameworks).

5 Conclusion

We have described a method, DC3, for fast approximate solutions to optimization problems with hard constraints. Our approach includes a neural network that outputs a partial set of variables, a differentiable completion procedure that fills in remaining variables according to equality constraints, and a differentiable correction procedure that fixes inequality violations. We find that DC3 yields solutions of significantly better feasibility and objective value than other approximate solvers on convex and non-convex optimization tasks.

We note that, while DC3 provides a general framework for tackling constrained optimization, depending on the setting, the expensiveness of both the completion and correction procedures may vary (e.g., implicit solutions may be more time-consuming, or gradient descent may converge more or less easily). We believe that, while our method as stated is broadly applicable, it will be possible in future work to design further improvements tailored to specific problem instances, for example by designing problem-dependent correction procedures.

REFERENCES

Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Brandon Amos. *Differentiable optimization-based modeling for machine learning*. PhD thesis, PhD thesis. Carnegie Mellon University, 2019.

Brandon Amos and J Zico Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning (ICML)*, 2017.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Kyri Baker. Learning warm-start points for AC optimal power flow. In *2019 IEEE 29th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6. IEEE, 2019.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

Tom Beucler, Stephan Rasp, Michael Pritchard, and Pierre Gentine. Achieving conservation of energy in neural network emulators for climate modeling. *arXiv preprint arXiv:1906.06622*, 2019.

Enzo Busseti, Walaa M Moursi, and Stephen Boyd. Solution refinement at regular points of conic problems. *Computational Optimization and Applications*, 74(3):627–643, 2019.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International Conference on Machine Learning (ICML)*, 2016.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Josip Djolonga and Andreas Krause. Differentiable learning of submodular models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Priya L Donti, Brandon Amos, and J Zico Kolter. Task-based end-to-end model learning in stochastic optimization. *Preprint arXiv:1703.04529*, 2017.

Thomas Frerix, Matthias Nießner, and Daniel Cremers. Homogeneous linear inequality constraints for neural network activations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 748–749, 2020.

Stephen Gould, Richard Hartley, and Dylan Campbell. Deep declarative networks: A new hope. *Preprint arXiv:1909.04866*, 2019.

Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In *Advances in Neural Information Processing Systems*, pp. 15379–15389, 2019.

Fouad Hasan, Amin Kargarian, and Ali Mohammadi. A survey on applications of machine learning for optimal power flow. In *2020 IEEE Texas Power and Energy Conference (TPEC)*, pp. 1–6. IEEE, 2020.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, 2015.

Slawomir Koziel and Leifur Leifsson. *Surrogate-based modeling and optimization*. Springer, 2013.

Jason D Lee, Ioannis Panageas, Georgios Piliouras, Max Simchowitz, Michael I Jordan, and Benjamin Recht. First-order methods almost always avoid saddle points. *Preprint arXiv:1710.07406*, 2017.

Chun Kai Ling, Fei Fang, and J Zico Kolter. What game are we playing? End-to-end learning in normal and extensive form games. *Preprint arXiv:1805.02777*, 2018.

Sidhant Misra, Line Roald, and Yeesian Ng. Learning for constrained optimization: Identifying optimal active constraint sets. *arXiv preprint arXiv:1802.09639*, 2018.

Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pp. 8026–8037, 2019.

David Rolnick, Priya L Donti, Lynn H Kaack, Kelly Kochanski, Alexandre Lacoste, Kris Sankaran, Andrew Slavin Ross, Nikola Milojevic-Dupont, Natasha Jaques, Anna Waldman-Brown, et al. Tackling climate change with machine learning. *Preprint arXiv:1906.05433*, 2019.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020. doi: 10.1007/s12532-020-00179-2. URL https://doi.org/10.1007/s12532-020-00179-2.

Sebastian Tschiatschek, Aytunc Sahin, and Andreas Krause. Differentiable submodular maximization. *Preprint arXiv:1803.01785*, 2018.

Po-Wei Wang, Priya L Donti, Bryan Wilder, and Zico Kolter. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning (ICML)*, 2019.

Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *AAAI Conference on Artificial Intelligence*, 2018.

Ahmed Zamzam and Kyri Baker. Learning optimal solutions for extremely fast AC optimal power flow. *Preprint arXiv:1910.01213*, 2019.

Ray D Zimmerman, Carlos E Murillo-Sánchez, and Deqiang Gan. MATPOWER: A MATLAB power system simulation package. *Manual, Power Systems Engineering Research Center, Ithaca NY*, 1, 1997.

A ADDITIONAL QP TASK RESULTS

		10	30	50	70	90
Optimizers (osqp, qpth)	Obj. val.	-17.33 (0.00)	-16.33 (0.00)	-15.05 (0.00)	-14.61 (0.00)	-14.26 (0.00)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
DC3	Obj. val.	-15.97 (0.31)	-14.41 (0.33)	-13.85 (0.02)	-12.09 (0.08)	-11.86 (0.09)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
DC3, $\neq$	Obj. val.	-15.69 (0.02)	-14.10 (0.03)	-12.55 (0.02)	-12.11 (0.03)	-11.71 (0.02)
	Max eq.	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
DC3, $\not\leq$ train	Obj. val.	-15.03 (0.60)	-13.87 (0.10)	-1.15 (1.12)	-11.08 (0.08)	-11.78 (0.06)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.01 (0.01)	0.00 (0.00)	0.00 (0.00)
DC3, $\not\leq$ train/test	Obj. val.	-15.03 (0.60)	-13.87 (0.10)	-0.65 (1.86)	-11.08 (0.08)	-11.78 (0.06)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.22 (0.27)	0.00 (0.00)	0.00 (0.00)
DC3, no soft loss	Obj. val.	-21.82 (0.01)	-21.85 (0.00)	-20.40 (0.00)	-21.83 (0.03)	-21.85 (0.00)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	23.81 (0.18)	23.91 (0.06)	14.15 (0.06)	23.70 (0.22)	23.80 (0.10)
NN	Obj. val.	-15.65 (0.03)	-14.09 (0.05)	-12.56 (0.03)	-12.12 (0.04)	-11.70 (0.03)
	Max eq.	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
NN, $\leq$ test	Obj. val.	-15.65 (0.03)	-14.09 (0.05)	-12.56 (0.03)	-12.12 (0.04)	-11.70 (0.03)
	Max eq.	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)	0.35 (0.00)
	Max ineq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
Eq. NN	Obj. val.	-16.97 (0.18)	-15.48 (0.10)	-9.51 (0.30)	-14.21 (0.06)	-14.10 (0.06)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	1.52 (0.25)	1.76 (0.05)	8.90 (0.88)	2.24 (0.03)	1.26 (0.02)
Eq. NN, $\leq$ test	Obj. val.	-16.91 (0.13)	-16.08 (0.05)	-14.90 (0.01)	-14.49 (0.05)	-14.11 (0.04)
	Max eq.	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	Max ineq.	0.78 (0.05)	0.84 (0.03)	0.59 (0.03)	0.92 (0.04)	0.98 (0.00)

Table A.1: Results on QP task for 100 variables and 50 equality constraints. The number of *inequality* constraints varies as $10, 30, 50, 70, 90$. Content and interpretation as in Table 2.

B DETAILS ON HYPERPARAMETER TUNING

The following parameters were kept fixed for all neural network-based methods across all experiments, based on a small amount of informal experimentation to ensure training was stable and properly converged:

- Epochs: 1000
- Batch size:[3] 200
- Hidden layer size: 200 (for both hidden layers)
- Correction procedure stopping tolerance: 10^{-4}
- Correction procedure momentum: 0.5

For all remaining parameters, we performed hyperparameter tuning via a coordinate search over relevant parameters. Central values for this coordinate search were determined via a small amount of informal experimentation to ensure training was stable on that central set of parameters. Final parameters were chosen so as to prioritize feasibility (as determined via the mean and max violations of equality and inequality constraints), followed by objective value.

[3]While this batch size was used for training, final timing experiments were run with all test datapoints in one batch.

B.1 QUADRATIC PROGRAMS

We list the ranges over which different hyperparameters were tuned for each method, with central values for the coordinate search in *italics*, and the final parameter values in **bold**. In order to minimize the amount of tuning, we employ the hyperparameters obtained for DC3 to "DC3, $\not\leq$ train" and "DC3, no soft loss" as well (rather than tuning the latter two methods separately). For this set of experiments, we set the learning rate of the gradient-based correction procedure ρ_x to 10^{-7} for all methods, as most methods went unstable for higher correction learning rates.

	DC3 DC3, $\not\leq$ train DC3, no soft loss	DC3, $\neq$	NN	Eq. NN
Lr	10^{-2}, $\mathit{10^{-3}}$, $\mathbf{10^{-4}}$	10^{-2}, $\mathit{10^{-3}}$, $\mathbf{10^{-4}}$	10^{-2}, $\mathit{10^{-3}}$, $\mathbf{10^{-4}}$	10^{-2}, $\mathit{10^{-3}}$, $\mathbf{10^{-4}}$
$\lambda_g + \lambda_h$	1,***10***,100	1,*10*,**100**	1,*10*,**100**	–
$\frac{\lambda_h}{\lambda_g+\lambda_h}$	***0.5***, 0.1, 0.01	***0.5***, 0.9, 0.99	***0.5***, 0.9, 0.99	–
t_{test}	5, 10, 50, **100**, *1000*	5, 10, 50, **100**, *1000*	5, 10, 50, **100**, *1000*	5, 10, 50, **100**, *1000*
t_{train}	1, 2, *5*, 10, **100**	1, 2, *5*, 10, **100**	–	–

B.2 AC OPTIMAL POWER FLOW

We list the ranges over which different hyperparameters were tuned for each method, with central values for the coordinate search in *italics*, and the final parameter values in **bold**. As in the previous setting, in order to minimize the amount of tuning, we employ the hyperparameters obtained for DC3 to "DC3, $\not\leq$ train" and "DC3, no soft loss" as well (rather than tuning the latter two methods separately).

	DC3 DC3, $\not\leq$ train DC3, no soft loss	DC3, $\neq$	NN	Eq. NN
Lr	10^{-2}, $\boldsymbol{10^{-3}}$, 10^{-4}	10^{-2}, $\boldsymbol{10^{-3}}$, 10^{-4}	10^{-2}, $\boldsymbol{10^{-3}}$, 10^{-4}	10^{-2}, $\boldsymbol{10^{-3}}$, 10^{-4}
$\lambda_g + \lambda_h$	1,***10***,100	1,*10*,**100**	1,*10*,**100**	–
$\frac{\lambda_h}{\lambda_g+\lambda_h}$	***0.5***, 0.1, 0.01	***0.5***, 0.9, 0.99	***0.5***, 0.9, 0.99	–
t_{test}	5, *10*, 50, **100**	5, *10*, 50, **100**	5, *10*, 50, **100**	5, *10*, 50, **100**, 200
t_{train}	1, 2, ***5***, 10, 100	1, 2, ***5***, 10, 100	–	–
Lr, ρ_x	10^{-3}, $\boldsymbol{10^{-4}}$, 10^{-5}	10^{-4}, $\boldsymbol{10^{-5}}$, 10^{-6}	$\mathbf{10^{-5}}$, $\mathit{10^{-6}}$, 10^{-7}	$\mathbf{10^{-5}}$, $\mathit{10^{-6}}$, 10^{-7}

C DETAILS OF DC3 FOR ACOPF

C.1 PROBLEM SETTING

We consider the problem of ACOPF defined in §4.2.

Let $\mathcal{B}$ denote the overall set of buses (i.e., nodes in the power network). In any instance of ACOPF there exists a set $\mathcal{D} \subseteq \mathcal{B}$ of *load (demand) buses* at which p_g and q_g are identically zero, as well as a set $\mathcal{R} \subseteq \mathcal{B}$ of *reference (slack) buses* at which p_g and q_g are potentially nonzero, and where the voltage angle $\angle v$ is known. Let $\mathcal{G} = \mathcal{B} \setminus (\mathcal{D} \cup \mathcal{R})$ be the remaining *generator buses* at which p_g and q_g are potentially nonzero, but where the voltage angle $\angle v$ is not known.

Then, we may rewrite Equation (6) as follows, where $v \equiv |v|e^{i\angle v}$ and $W \equiv W_r + W_i i$:

$$\begin{aligned}
\underset{p_g, q_g, |v|, \angle v \in \mathbb{R}^{|\mathcal{B}|}}{\text{minimize}} \quad & p_g^T A p_g + b^T p_g && \text{(C.1a)}\\
\text{subject to} \quad & p_g^{\min} \leq p_g \leq p_g^{\max} && \text{(C.1b)}\\
& q_g^{\min} \leq q_g \leq q_g^{\max} && \text{(C.1c)}\\
& v^{\min} \leq |v| \leq v^{\max} && \text{(C.1d)}\\
& (\angle v)_{\mathcal{R}} = \phi_{\text{slack}} && \text{(C.1e)}\\
& (p_g)_{\mathcal{D}} = (q_g)_{\mathcal{D}} = 0 && \text{(C.1f)}\\
& p_g - p_d - \text{diag}(v_r)(W_r v_r - W_i v_i) - \text{diag}(v_i)(W_i v_r + W_r v_i) = 0 && \text{(C.1g)}\\
& q_g - q_d + \text{diag}(v_r)(W_i v_r + W_r v_i) - \text{diag}(v_i)(W_r v_r - W_i v_i) = 0 && \text{(C.1h)}\\
& \quad \text{where } v_r = |v|\cos(\angle v) \text{ and } v_i = |v|\sin(\angle v)
\end{aligned}$$

(While we write the problem in this form to minimize notation, in practice, some of the constraints in the above problem – e.g., (C.1b), (C.1c), and (C.1f) – can be condensed.)

C.2 Overall approach

As pointed out in Zamzam & Baker (2019), given p_d, q_d, $(p_g)_{\mathcal{G}}$ and $|v|_{\mathcal{B}\setminus\mathcal{D}}$, the remaining variables $(p_g)_{\mathcal{R}}$, $(q_g)_{\mathcal{B}\setminus\mathcal{D}}$, $|v|_{\mathcal{D}}$, and $(\angle v)_{\mathcal{B}\setminus\mathcal{R}}$ can be recovered via the power flow equations (C.1g)–(C.1h).

As such, our implementation of DC3 may be outlined as follows.

- **Input:** $x = \begin{bmatrix} p_d^T & q_d^T \end{bmatrix}^T$. (The constant $\angle v_{\mathcal{R}}$ is fixed across problem instances.)
- **Neural network:** Output $\alpha \in [0,1]^{|\mathcal{G}|}$ and $\beta \in [0,1]^{|\mathcal{B}|-|\mathcal{D}|}$ (applying a sigmoid to the final layer of the network), and compute:
$$\begin{aligned}
(p_g)_{\mathcal{G}} &= \alpha(p_g^{\min})_{\mathcal{G}} + (1-\alpha)(p_g^{\max})_{\mathcal{G}},\\
|v|_{\mathcal{B}\setminus\mathcal{D}} &= \beta v_{\mathcal{B}\setminus\mathcal{D}}^{\min} + (1-\beta) v_{\mathcal{B}\setminus\mathcal{D}}^{\max}.
\end{aligned}$$
(In other words, output a partial set of variables, and enforce box constraints on this set of variables via sigmoids in the neural network.)
- **Completion procedure:** Given $z = \begin{bmatrix} (p_g)_{\mathcal{G}}^T & |v|_{\mathcal{B}\setminus\mathcal{D}}^T \end{bmatrix}^T$, solve Equations (C.1g)–(C.1h) for the remaining quantities $(p_g)_{\mathcal{R}}$, $(q_g)_{\mathcal{B}\setminus\mathcal{D}}$, $|v|_{\mathcal{D}}$, and $(\angle v)_{\mathcal{B}\setminus\mathcal{R}}$ as described below. Output all decision variables $y = \begin{bmatrix} (p_g)_{\mathcal{G}}^T & (q_g)_{\mathcal{G}}^T & |v|^T & (\angle v)^T \end{bmatrix}^T$.
- **Correction procedure:** Correct y using the gradient-based feasibility correction procedure described in §3.2.
- **Backpropagate:** Compute the loss and backpropagate as described below to update neural network parameters. Repeat until convergence.

We now describe the forward and backward passes through the completion procedure, where we have introduced several "tricks" that significantly reduce the computational cost, including some that are specific to the ACOPF setting.

For ease of notation throughout, we will let $J \in \mathbb{R}^{1+2|\mathcal{B}|}$ denote the Jacobian of the equality constraints (C.1e)–(C.1h) with respect to the complete vector y of decision variables. This matrix, which we pre-compute, will be useful throughout our computations.

C.3 Solving the completion

The neural network outputs $(p_g)_{\mathcal{G}}$ and $|v|_{\mathcal{B}\setminus\mathcal{D}}$ are the inputs to the completion procedure. Theoretically, we could use Newton's method to solve for all additional variables from the equality constraints. However, in practice we find that solving for all variables using Newton's method is sometimes unstable. Fortunately, we can divide the completion procedure into two substeps, where Step 1 invokes Newton's method to identify some of the variables, while Step 2 solves in closed form for the others. This greatly improves the stability of the completion procedure.

Step 1. Compute $|v|_{\mathcal{D}}$, $(\angle v)_{\mathcal{B}\setminus\mathcal{R}}$ via Newton's method using real power flow constraints (C.1g) at buses $\mathcal{B}\setminus\mathcal{R}$, and reactive power constraints (C.1h) at buses $\mathcal{D}$ (note that the number of equations matches the number of variables being identified). We initialize Newton's method by fixing variables determined by the neural network and those already determined in Step 1 of the completion process, and initializing $|v|_{\mathcal{D}}$, $(\angle v)_{\mathcal{B}\setminus\mathcal{R}}$ at generic initial values (these are provided in the ACOPF task setup and are typically used to initialize state-of-the-art solvers). Note that the remaining variables, those determined in Step 2 of the completion process, do not actually appear in the relevant equality constraints and therefore do not need to be set.

Let $J_{\text{Step 1}}$ denote the submatrix of J corresponding to the equality constraints (C.1g)$_{\mathcal{B}\setminus\mathcal{R}}$ and (C.1h)$_{\mathcal{D}}$ and the voltage variables $|v|_{\mathcal{D}}$ and $(\angle v)_{\mathcal{B}\setminus\mathcal{R}}$ that we are solving in this step. At the tth step of Newton's method, let $h_{\text{Step 1}}(t) \in \mathbb{R}^{|\mathcal{B}|-|\mathcal{R}|+|\mathcal{D}|}$ denote the vector of values on the left-hand side of the relevant equality constraints (which we wish to equal identically zero), evaluated at the current setting of all problem variables. Our Newton's method updates are then

$$\begin{bmatrix} |v|_{\mathcal{D}} \\ (\angle v)_{\mathcal{B}\setminus\mathcal{R}} \end{bmatrix}_{t+1} = \begin{bmatrix} |v|_{\mathcal{D}} \\ (\angle v)_{\mathcal{B}\setminus\mathcal{R}} \end{bmatrix}_{t} - J_{\text{Step 1}}^{-1}\, h_{\text{Step 1}}(t). \tag{C.2}$$

Step 2. Compute the remaining variables $(p_g)_{\mathcal{R}}$ and $(q_g)_{\mathcal{B}\setminus\mathcal{D}}$ via the remaining equality constraints – that is, the real power flow equations (C.1g) at buses $\mathcal{R}$ and the reactive power flow questions (C.1h) at buses in $\mathcal{B}\setminus\mathcal{D}$.

Steps 1 and 2 together allow us to complete all the decision variables.

C.4 Backpropagating through the completion

Let z denote the input to the completion procedure and let z_1 and z_2 denote the variables respectively derived during Steps 1 and 2 of completion. That is:

$$z \equiv \begin{bmatrix} (p_g)_{\mathcal{G}} \\ |v|_{\mathcal{B}\setminus\mathcal{D}} \end{bmatrix}, \qquad z_1 \equiv \begin{bmatrix} |v|_{\mathcal{D}} \\ (\angle v)_{\mathcal{B}\setminus\mathcal{R}} \end{bmatrix}, \qquad z_2 \equiv \begin{bmatrix} (p_g)_{\mathcal{R}} \\ (q_g)_{\mathcal{B}\setminus\mathcal{D}} \end{bmatrix}.$$

Then, we wish to backpropagate gradients of an arbitrary loss function $\ell(x, z, z_1, z_2)$, both in order to train our neural network and to perform our gradient-based correction procedure. That is, we must compute the total derivative $\mathrm{d}\ell/\mathrm{d}z$ given the partial derivatives $\partial\ell/\partial z, \partial\ell/\partial z_1$, and $\partial\ell/\partial z_2$.

Applying the chain rule through the two steps of the completion procedure, we have:

$$\frac{\mathrm{d}\ell}{\mathrm{d}z} = \frac{\partial\ell}{\partial z} + \frac{\partial\ell}{\partial z_1}\frac{\partial z_1}{\partial z} + \frac{\partial\ell}{\partial z_2}\frac{\partial z_2}{\partial z} + \frac{\partial\ell}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial z}. \tag{C.3}$$

We now consider each of these terms in this equality, except for $\partial\ell/\partial z, \partial\ell/\partial z_1, \partial\ell/\partial z_2$, which we may assume have already been computed.

Step 2. Let $J_{\text{Step 2}}$ denote the submatrix of J corresponding to the partial derivatives of the equality constraints used in Step 2 with respect to the voltage variables $|v|^T, (\angle v)^T$. Then, we have:

$$\frac{\partial \begin{bmatrix} (p_g)_{\mathcal{R}}^T & (q_g)_{\mathcal{B}\setminus\mathcal{D}}^T \end{bmatrix}^T}{\partial \begin{bmatrix} |v|^T & (\angle v)^T \end{bmatrix}^T} = -J_{\text{Step 2}}. \tag{C.4}$$

As $\frac{\partial (q_g)_{\mathcal{B}\setminus\mathcal{D}}}{\partial (p_g)_{\mathcal{G}}} = 0$, this gives us both the terms $\partial z_2/\partial z_1, \partial z_2/\partial z$ in (C.3).

Step 1. Consider the total differential through the equality constraints (C.1g)$_{\mathcal{B}\setminus\mathcal{R}}$ and (C.1h)$_{\mathcal{D}}$, where all terms besides W_r and W_i are viewed as parameters through which to differentiate. We rearrange this total differential to put differentials of input quantities to Step 1 on one side and differentials of outputs from Step 1 on the other side:

$$J_{\text{Step 1}} \begin{bmatrix} \mathrm{d}(|v|)_{\mathcal{D}} \\ \mathrm{d}(\angle v)_{\mathcal{B}\setminus\mathcal{R}} \end{bmatrix} = \begin{bmatrix} -\mathrm{d}(p_g)_{\mathcal{B}\setminus\mathcal{R}} + \mathrm{d}(p_d)_{\mathcal{B}\setminus\mathcal{R}} \\ \mathrm{d}(q_d)_{\mathcal{D}} \end{bmatrix} - J_{\text{Step 1b}} \begin{bmatrix} \mathrm{d}|v|_{\mathcal{B}\setminus\mathcal{D}} \\ \mathrm{d}(\angle v)_{\mathcal{R}} \end{bmatrix}, \tag{C.5}$$

where $J_{\text{Step 1}}$ is as in the forward pass of Step 1, and $J_{\text{Step 1b}}$ denotes the submatrix of J corresponding to the equality constraints (C.1g)$_{\mathcal{B}\setminus\mathcal{R}}$ and (C.1h)$_{\mathcal{D}}$ and the voltage variables $(|v|)_{\mathcal{B}\setminus\mathcal{D}}, (\angle v)_{\mathcal{R}}$.

While we could compute $\begin{bmatrix} \mathrm{d}|v|_{\mathcal{D}} \\ \mathrm{d}(\angle v)_{\mathcal{B}\setminus\mathcal{R}} \end{bmatrix}$ explicitly, we can improve efficiency by not computing and storing a large intermediate Jacobian explicitly. Instead, we can directly compute what we need, which is the product of this Jacobian with matrices of smaller dimensions. Define

$$K \equiv \left(\frac{\partial \ell}{\partial z_1} + \frac{\partial \ell}{\partial z_2} \frac{\partial z_2}{\partial z_1} \right) J_{\text{Step 1}}^{-1},$$

using our computation (C.4) above to evaluate $\partial z_2 / \partial z_1$. Note furthermore that $J_{\text{Step 1}}^{-1}$ was already computed in the forward pass, meaning that we do not need to perform an additional matrix inversion.

Now, we can use (C.5) to derive:

$$\left(\frac{\partial \ell}{\partial z_1} + \frac{\partial \ell}{\partial z_2} \frac{\partial z_2}{\partial z_1} \right) \mathrm{d}z_1 = K \left(\begin{bmatrix} -\mathrm{d}(p_g)_{\mathcal{B}\setminus\mathcal{R}} + \mathrm{d}(p_d)_{\mathcal{B}\setminus\mathcal{R}} \\ \mathrm{d}(q_d)_{\mathcal{D}} \end{bmatrix} - J_{\text{Step 1b}} \begin{bmatrix} \mathrm{d}(|v|)_{\mathcal{B}\setminus\mathcal{D}} \\ \mathrm{d}(\angle v)_{\mathcal{R}} \end{bmatrix} \right),$$

which gives us

$$\frac{\partial \ell}{\partial z_1} \frac{\partial z_1}{\partial z} + \frac{\partial \ell}{\partial z_2} \frac{\partial z_2}{\partial z_1} \frac{\partial z_1}{\partial z}.$$

Putting it all together.

Combining our logic described above for Steps 2 and 1, we can recover all terms in (C.3) and therefore identify $\mathrm{d}\ell / \mathrm{d}z$.