Online Representation Learning on the Open Web

Ellis L. Brown, II

CMU-CS-23-1XX
May 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee

Deepak Pathak	*Carnegie Mellon University*
Deva Ramanan	*Carnegie Mellon University*
Alexei A. Efros	*University of California, Berkeley*

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Keywords: machine learning, deep learning, computer vision, representation learning, self-supervised learning, active learning, online learning, internet

To my parents, John and Amy, for their unwavering support and love, and to Mara for putting up with all of the madness.

Abstract

Modern vision models typically rely on fine-tuning general-purpose models pre-trained on large, static datasets. These general-purpose models only capture the knowledge within their pre-training datasets, which are tiny, out-of-date snapshots of the Internet—where *billions* of images are uploaded daily.

We suggest in this thesis an alternate approach: rather than hoping our static datasets transfer to our desired tasks after large-scale pre-training, we propose dynamically utilizing the Internet to quickly train a small-scale model that does extremely well on the task at hand. Our approach, called **Internet Explorer**, explores the web in a self-supervised manner to progressively find relevant examples that improve performance on a desired target dataset. It cycles between searching for images on the Internet with text queries, self-supervised training on downloaded images, determining which images were useful, and prioritizing what to search for next.

We evaluate **Internet Explorer** across several datasets and show that it outperforms or matches CLIP oracle performance by using just a single GPU desktop to actively query the Internet for 30–40 hours.

The source code for this thesis document is available in open source form at:
https://github.com/ellisbrown/cmu-masters-thesis

Acknowledgments

TODO

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The field of machine learning has grown rapidly over the past few years and has a growing set of well-defined and well-understood operations and paradigms that allow a practitioner to inject domain knowledge into the modeling procedure. These operations include linear maps, convolutions, activation functions, random sampling, and simple projections (e.g. onto the simplex or Birkhoff polytope). In addition to these layers, the practitioner can also inject domain knowledge at a higher-level of how modeling components interact. Paradigms are becoming well-established for modeling images, videos, audio, sequences, permutations, and graphs, among others. This thesis proposes a new set of primitive operations and paradigms based on optimization that allow the practitioner to inject specialized domain knowledge into the modeling procedure.

Optimization plays a large role in machine learning for parameter optimization or architecture search. In this thesis, we argue that optimization should have a third role in machine learning separate from these two, that it can be used as a modeling tool inside of the inference procedure. Optimization is a powerful modeling tool and as we show in Section 2.4.4, many of the standard operations such as the ReLU, sigmoid, and softmax can all be interpreted as explicit closed-form solutions to constrained convex optimization problems. We also highlight in Section 2.2.1 that the standard feedforward supervised learning setup can be captured by an energy-based optimization problem. Thus these techniques are captured as special cases of the general optimization-based modeling methods we study in this thesis that don't necessarily have explicit closed-form solutions. This generalization adds new modeling capabilities that were not possible before and enables new ways that practitioners can inject domain knowledge into the models.

From an optimization viewpoint, the techniques we propose in this thesis can be used for partial modeling of optimization problems. Traditionally a modeler needs to have a complete analytic view of their system if they want to use optimization to solve their problem, such as in many control, planning, and scheduling tasks. The techniques in this thesis lets the practitioner leave latent parts in their optimization-based modeling procedure that can then be learned from data.

1.1 Summary of research contributions

The first portion of this thesis presents foundational modeling techniques that use optimization-based modeling:

- Chapter 3 presents the *OptNet* architecture that shows how to use constrained convex optimization as a layer in an end-to-end architecture.
  - Section 3.3 presents the formulation of these architectures and shows how back-propagation can be done in them by implicitly differentiating the KKT conditions.
  - Section 3.3.2 studies the representational power of these architectures and proves how they can represent any piecewise linear function including the ReLU.
  - Section 3.3.1 presents our efficient QP solver for these layers and Section 3.3.1 shows how we can compute the backwards pass with almost no computational overhead.
  - Section 3.4 shows empirical results that uses OptNet for a synthetic denoising task and to learn the rules of the Sudoku game.
- Chapter 4 presents the *input-convex neural network* architecture.
  - Section 4.4 discusses efficient inference techniques for these architectures. We propose a new inference technique called the Bundle-Entropy method in Section 4.4.4.
  - Section 4.5 discusses efficient learning techniques for these architecture.
  - Section 4.6 shows empirical results applying ICNNs to structured prediction, data imputation, and continuous-action Q learning.

The remaining portions discuss applications and extensions of OptNet.

- **??** presents our *differentiable model predictive control* (MPC) work as a step towards leveraging MPC as a differentiable policy class for reinforcement learning in continuous state-action spaces.
  - **??** shows how to efficiently differentiate through the Linear Quadratic Regulator (LQR) by solving another LQR problem. This result comes from implicitly differentiating the KKT conditions of LQR and interpreting the resulting system as solving another LQR problem.
  - **??** shows how to differentiate through non-convex MPC problems by differentiating through the fixed point obtained when solving the MPC problem with sequential quadratic programming (SQP).
  - **??** shows our empirical results using imitation learning in the pendulum and cartpole environments. Notably, we show why doing end-to-end learning with a controller is important in tasks when the expert is non-realizable.

- ?? presents the Limited Multi-Layer Projection (LML) layer for top-k learning problems.
  - ?? introduces the LML projection problem that we study.
  - ?? shows how to efficiently solve the LML projection problem by solving the dual with a parallel bracketed root-finding method.
  - ?? presents how to maximize the top-k recall with the LML layer.
  - ?? shows our empirical results on top-k image classification and scene graph generation.
- Chapter 5 shows how to make differentiable `cvxpy` optimization layers by differentiating through the internal transformations and internal cone program solver. This enables rapid prototyping of all of the convex optimization-based modeling methods we consider in this thesis.
  - Section 5.3 shows how to differentiate cone programs (including non-polyhedral cone programs) by implicitly differentiating the residual map of Minty's parameterization of the homogeneous self-dual embedding.
  - Section 5.5 shows examples of using our package to implement optimization layers for the ReLU, sigmoid, softmax; projections onto polyhedral and ellipsoidal sets; and the OptNet QP.

1.2 Summary of open source contributions

The code and experiments developed for this thesis are free and open-source:

- https://github.com/locuslab/icnn: TensorFlow experiments for the input-convex neural networks work presented in Chapter 4.
- https://locuslab.github.io/qpth/ and https://github.com/locuslab/qpth: A stand-alone PyTorch library for the OptNet QP layers presented in Chapter 3.
- https://github.com/locuslab/optnet: PyTorch experiments for the OptNet work presented in Chapter 3.
- https://locuslab.github.io/mpc.pytorch and https://github.com/locuslab/mpc.pytorch: A stand-alone PyTorch library for the differentiable model predictive control approach presented in **??**.
- https://github.com/locuslab/differentiable-mpc: PyTorch experiments for the differentiable MPC work presented in **??**.

I have also created the following open source projects during my Ph.D.:

- https://github.com/bamos/block: An intelligent block matrix library for numpy, PyTorch, and beyond.
- https://github.com/bamos/dcgan-completion.tensorflow: Image Completion with Deep Learning in TensorFlow.
- https://github.com/cmusatyalab/openface: Face recognition with deep neural networks.
- https://github.com/bamos/densenet.pytorch: A PyTorch implementation of DenseNet.

1.3 Summary of publications

The content of Chapter 3 appears in:

> [AK17] Brandon Amos and J. Zico Kolter. "OptNet: Differentiable Optimization as a Layer in Neural Networks". In: *Proceedings of the International Conference on Machine Learning*. 2017

The content of Chapter 4 appears in:

> [AXK17] Brandon Amos, Lei Xu, and J. Zico Kolter. "Input Convex Neural Networks". In: *Proceedings of the International Conference on Machine Learning*. 2017

The content of **??** appears in:

> [Amo+18b] Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J. Zico Kolter. "Differentiable MPC for End-to-end Planning and Control". In: *Advances in Neural Information Processing Systems*. 2018, pp. 8299–8310

Non-thesis research: I have also pursued the following research directions during my Ph.D. studies. These are excluded from the remainder of this thesis.

> [Amo+18a] Brandon Amos, Laurent Dinh, Serkan Cabi, Thomas Rothörl, Sergio Gómez Colmenarejo, Alistair Muldal, Tom Erez, Yuval Tassa, Nando de Freitas, and Misha Denil. "Learning Awareness Models". In: *International Conference on Learning Representations*. 2018
>
> [ALS16] Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. *OpenFace: A general-purpose face recognition library with mobile applications*. Tech. rep. Technical Report CMU-CS-16-118, CMU School of Computer Science, 2016
> **Available online at:** https://cmusatyalab.github.io/openface

I have also contributed to the following publications as a non-primary author.

> Priya L Donti, Brandon Amos, and J. Zico Kolter. "Task-based End-to-end Model Learning". In: *NIPS*. 2017
>
> Han Zhao, Tameem Adel, Geoff Gordon, and Brandon Amos. "Collapsed Variational Inference for Sum-Product Networks". In: *ICML*. 2016

Zhuo Chen et al. "An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM. 2017, p. 12

Zhuo Chen, Lu Jiang, Wenlu Hu, Kiryong Ha, Brandon Amos, Padmanabhan Pillai, Alex Hauptmann, and Mahadev Satyanarayanan. "Early Implementation Experience with Wearable Cognitive Assistance Applications". In: *WearSys*. 2015

Nigel Andrew Justin Davies, Nina Taft, Mahadev Satyanarayanan, Sarah Clinch, and Brandon Amos. "Privacy mediators: helping IoT cross the chasm". In: *HotMobile*. 2016

Junjue Wang, Brandon Amos, Anupam Das, Padmanabhan Pillai, Norman Sadeh, and Mahadev Satyanarayanan. "A Scalable and Privacy-Aware IoT Service for Live Video Analytics". In: *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM. 2017, pp. 38–49

Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. "The Case for Offload Shaping". In: *HotMobile*. 2015

Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. "Edge Analytics in the Internet of Things". In: *IEEE Pervasive Computing* 2 (2015), pp. 24–31

Ying Gao, Wenlu Hu, Kiryong Ha, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. *Are Cloudlets Necessary?* Tech. rep. Technical Report CMU-CS-15-139, CMU School of Computer Science, 2015

Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. "You can teach elephants to dance: agile VM handoff for edge computing". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM. 2017, p. 12

Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. "Quantifying the impact of edge computing on mobile applications". In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM. 2016, p. 5

Chapter 2

Preliminaries and Background

This section provides a broad overview of foundational ideas and background material relevant to this thesis. In most chapters of this thesis, we include a deeper discussion of the related literature relevant to that material.

2.1 Preliminaries

The content in this thesis builds on the following topics. We assume preliminary knowledge of these topics and give a limited set of key references here. The reader should have an understanding of statistical and machine learning modeling paradigms as described in Wasserman [Was13], Bishop [Bis07], and Friedman, Hastie, and Tibshirani [FHT01]. Our contributions mostly focus on end-to-end modeling with deep architectures as described in Schmidhuber [Sch15] and Goodfellow, Bengio, Courville, and Bengio [Goo+16] with applications in computer vision as described in Forsyth and Ponce [FP03], Bishop [Bis07], and Szeliski [Sze10]. Our contributions also involve optimization theory and applications as described in Bertsekas [Ber99], Boyd and Vandenberghe [BV04], Bonnans and Shapiro [BS13], Griewank and Walther [GW08], Nocedal and Wright [NW06], Sra, Nowozin, and Wright [SNW12], and Wright [Wri97]. One application area of this thesis work focuses on control and reinforcement learning. Control is one kind of optimization-based modeling and is further described in Bertsekas, Bertsekas, Bertsekas, and Bertsekas [Ber+05], Sastry and Bodson [SB11], and Levine [Lev17b]. Reinforcement learning methods are summarized in Sutton, Barto, et al. [SB+98] and Levine [Lev17a].

2.2 Energy-based Learning

Energy-based learning is a machine learning method typically used in supervised settings that explicitly adds relationships and dependencies to the model's output space. This is in contrast to purely feed-forward models that typically cannot explicitly capture dependencies in the output space. At the core of energy-based learning methods is a scalar-valued *energy* function $E_\theta(x, y) : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ parameterized by θ that measures the fit between

some input x and output y. Inference in energy-based models is done by solving the optimization problem

$$\hat{y} = \operatorname*{argmin}_{y} E_\theta(x, y). \tag{2.1}$$

We note that this is a powerful formulation for modeling and learning and subsumes the representational capacity of standard deep feedforward models, which we show how to do in Section 2.2.1. The energy function can also be interpreted from a probabilistic lens as the negated unnormalized joint distribution over the input and output spaces.

Energy-based methods have been in use for over a decade and the tutorial LeCun, Chopra, Hadsell, Ranzato, and Huang [LeC+06] overviews many of the foundational methods and challenges in energy-based learning. The two main challenges for energy-based learning are 1) learning the parameters θ of the energy function E_θ and 2) efficiently solving the inference procedure in Equation (2.1). These challenges have historically been tamed by using simpler energy functions consisting of hand-engineered feature extractors for the inputs x and linear functions of y. This captures models such as Markov random fields [Li94] and conditional random fields [LMP01; SM+12]. Standard gradient-based methods are difficult to use for parameter learning because $\hat{y}$ depends on θ through the argmin operator, which is not always differentiable. Historically, a common approach to doing parameter learning in energy-based models has been to directly shape the energy function with a max-margin approach Taskar, Guestrin, and Koller [TGK04] and Taskar, Chatalbashev, Koller, and Guestrin [Tas+05].

More recently, there has been a strong push to further incorporate structured prediction methods like conditional random fields as the "last layer" of a deep network architecture [PBX09; Zhe+15; Che+15a] as well as in deeper energy-based architectures [BM16; BYM17; Bel17; WFU16]. We further discuss Structured Prediction Energy Networks (SPENs) in Section 2.2.2.

An ongoing discussion in the community argues whether adding the dependencies explicitly in an energy-based is useful or not. Feedforward models have a remarkable representational capacity that can implicitly learn the dependencies and relationships from data without needing to impose additional structure or modeling assumptions and without making the model more computationally expensive with an optimization-based inference procedure. One argument against this viewpoint that supports energy-based modeling is that explicitly including modeling information improves the data efficiency and requires less samples to learn because some structure and knowledge is already present in the model and does not have to be learned from scratch.

2.2.1 Energy-based Models Subsume Feedforward Models

We highlight the power of energy-based modeling for supervised learning by noting how they subsume deep feedforward models. Let $\hat{y} = f_\theta(x)$ be a deep feedforward model. The energy-based representation of this model is $E(x, y) = ||y - f_\theta(x)||_2^2$ and inference becomes the convex optimization problem $\hat{y} = \operatorname{argmin}_y E(x, y)$, which has the exact solution $\hat{y} = f_\theta(x)$. An energy function that has more structure over the output space adds representational capacity that a feedforward model wouldn't be able to capture explicitly.

2.2.2 Structured Prediction Energy Networks

Structured Prediction Energy Networks (SPENs) [BM16; BYM17; Bel17] are a way of bridging the gap between modern deep learning methods and classical energy-based learning methods. SPENs provide a deep structure over input and output spaces by representing the energy function $E_\theta(x, y)$ with a standard feed-forward neural network. This expressive formulation comes at the cost of making the inference procedure in Equation (2.1) difficult and non-convex. SPENs typically use an approximate inference procedure by taking a fixed-number of gradient descent steps for inference. For learning, SPENs typically replace the inference with an *unrolled gradient-based optimizer* that starts with some prediction y_0 and takes a fixed number of gradient steps to minimize the energy function

$$y_{i+1} = y_i - \alpha \nabla_y E_\theta(x, y_i).$$

The final iterate as then taken as the prediction $\hat{y} \triangleq y_N$. Gradient-based parameter learning can be done by differentiating the prediction $\hat{y}$ with respect to θ by unrolling the inference procedure. Unrolling the inference procedure can be done in most autodiff frameworks such as PyTorch [Pas+17b] or TensorFlow [Aba+16]. activation functions with smooth first derivatives such as the sigmoid or softplus [GBB11] should be used to avoid discontinuities because unrolling the inference procedure involves computing $\nabla_\theta \nabla_y E_\theta(x, y)$.

2.3 Modeling with Domain-Specific Knowledge

The role of domain-specific knowledge in the machine learning and computer vision fields has been an active discussion topic over the past decade and beyond. Historically, domain knowledge such as fixed hand-crafted feature and edge detectors were rigidly part of the computer vision pipeline and have been overtaken by learnable convolutional models LeCun, Cortes, and Burges [LCB98] and Krizhevsky, Sutskever, and Hinton [KSH12]. To highlight the power of convolutional architectures, they provide a reasonable prior for vision tasks even without learning [UVL18]. Machine learning models extend far beyond the reach of vision tasks and the community has a growing interest on domain-specific priors rather than just using fully-connected architectures. These priors ideally can be integrated as end-to-end learnable modules into a larger system that are learned as a whole with gradient-based information. In contrast to pure fully-connected architectures, specialized submodules ideally improve the data efficiency of the model, add interpretability, and enable grey-box verification.

Recent work has gone far beyond the classic examples of adding modeling priors by using convolutional or sequential models. A full discussion of all of the recent improvements is beyond the scope of this thesis, and here we highlight a few key recent developments.

- Differentiable beam search [Goy+18] and differentiable dynamic programming [MB18]
- Differentiable protein simulator [Ing+18]
- Differentiable particle filters [JRB18]
- Neural ordinary differential equations [Che+18] and applications to reversible generative models [Gra+18]

- Relational reasoning on sets, graphs, and trees [Bat+18; Zah+17; KW16; Gil+17; San+17; HYL17; Bat+16; Xu+18; Far+17; She+18]
- Geometry-based priors [Bro+17; Gul+18; Mon+17; TT18; Li+18]
- Memory [SWF+15; GWD14; Gra+16; XMS16; Hil+15; PS17]
- Attention [BCB14; Vas+17; Wan+18]
- Capsule networks [SFH17; HSF18; XC18]
- Program synthesis [RD15; NLS15; Bal+16; Dev+17; Par+16]

2.4 Optimization-based Modeling

Optimization can be used for modeling in machine learning. Among many other applications, these architectures are well-studied for generic classification and structured prediction tasks [Goo+13; SRE11; BSS13; LeC+06; BM16; BYM17]; in vision for tasks such as denoising [Tap+07; SR14] or edge-aware smoothing [BP16]. Diamond, Sitzmann, Heide, and Wetzstein [Dia+17] presents unrolled optimization with deep priors. Metz, Poole, Pfau, and Sohl-Dickstein [Met+16] uses unrolled optimization within a network to stabilize the convergence of generative adversarial networks [Goo+14]. Indeed, the general idea of solving restricted classes of optimization problem using neural networks goes back many decades [KC88; Lil+93], but has seen a number of advances in recent years. These models are often trained by one of the following four methods.

2.4.1 Explicit Differentiation

If an analytic solution to the argmin can be found, such as in an unconstrained quadratic minimization, the gradients can often also be computed analytically. This is done in Tappen, Liu, Adelson, and Freeman [Tap+07] and Schmidt and Roth [SR14]. We cannot use these methods for the constrained optimization problems we consider in this thesis because there are no known analytic solutions.

2.4.2 Unrolled Differentiation

The argmin operation over an unconstrained objective can be approximated by a first-order gradient-based method and unrolled. These architectures typically introduce an optimization procedure such as gradient descent into the inference procedure. This is done in Domke [Dom12], Belanger, Yang, and McCallum [BYM17], Metz, Poole, Pfau, and Sohl-Dickstein [Met+16], Goodfellow, Mirza, Courville, and Bengio [Goo+13], Stoyanov, Ropson, and Eisner [SRE11], Brakel, Stroobandt, and Schrauwen [BSS13], and Finn, Abbeel, and Levine [FAL17]. The optimization procedure is unrolled automatically or manually [Dom12] to obtain derivatives during training that incorporate the effects of these in-the-loop optimization procedures.

Given an unconstrained optimization problem with a parameterized objective

$$\operatorname*{argmin}_x f_\theta(x),$$

gradient descent starts at an initial value x_0 and takes steps

$$x_{i+1} = x_i - \alpha \nabla_x f_\theta(x).$$

For learning, the final iterate of this procedure x_N can be taken as the output and $\partial x_N / \partial \theta$ can be computed with automatic differentiation.

In all of these cases, the optimization problem is unconstrained and unrolling gradient descent is often easy to do. When constraints are added to the optimization problem, iterative algorithms often use a projection operator that may be difficult to unroll through and storing all of the intermediate iterates may become infeasible.

2.4.3 Implicit argmin differentiation

Most closely related to this thesis work, there have been several applications of the implicit function theorem to differentiating through constrained convex argmin operations. These methods typically parameterize an optimization problem's objective or constraints and then applies the *implicit function theorem* (Theorem 1) to optimality conditions of the optimization problem that implicitly define the solution, such as the *KKT conditions* [BV04, Section 5.5.3]. We will first review the implicit function theorem and KKT conditions and then discuss related work in this space.

Implicit function analysis [DR09] typically focuses on solving an equation $f(p, x) = 0$ for x as a function s of p, *i.e.* $x = s(p)$. *Implicit differentiation* considers how to differentiate the solution mapping with respect to the parameters, *i.e.* $\nabla_p s(p)$. The *implicit function theorem* used in standard calculus textbooks can be traced back to the lecture notes from 1877-1878 of Dini [Din77] and is presented in Dontchev and Rockafellar [DR09, Theorem 1.B.1] as follows.

Theorem 1 (Implicit function theorem)**.** *Let $f : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^n$ be continuously differentiable in a neighborhood of $(\bar{p}, \bar{x})$ and such that $f(\bar{p}, \bar{x}) = 0$, and let the partial Jacobian of f with respect to x at $(\bar{p}, \bar{x})$, namely $\nabla_x f(\bar{p}, \bar{x})$, be nonsingular. Then the solution mapping $S(p) = \{x \in \mathbb{R}^n \mid f(p, x) = 0\}$ has a single-valued localization s around $\bar{p}$ for $\bar{x}$ which is continuously differentiable in a neighborhood Q of $\bar{p}$ with Jacobian satisfying $\nabla s(p) = -\nabla_x f(p, s(p))^{-1} \nabla_p f(p, s(p))$ for every $p \in Q$.*

In addition to the content in this thesis, several other papers apply the implicit function theorem to differentiate through the argmin operators. This approach frequently comes up in bilevel optimization [Gou+16; KP13] and sensitivity analysis [Ber99; FI90; BB08; BS13]. [Bar18] is a note on applying the implicit function theorem to the KKT conditions of convex optimization problems and highlights assumptions behind the derivative being well-defined. Gould, Fernando, Cherian, Anderson, Santa Cruz, and Guo [Gou+16] describes general techniques for differentiation through optimization problems, but only describe the case of exact equality constraints rather than both equality and inequality

constraints (they add inequality constraints via a barrier function). Johnson, Duvenaud, Wiltschko, Adams, and Datta [Joh+16] performs implicit differentiation on (multi-)convex objectives with coordinate subspace constraints. The older work of Mairal, Bach, and Ponce [MBP12] considers argmin differentiation for a LASSO problem, derives specific rules for this case, and presents an efficient algorithm based upon our ability to solve the LASSO problem efficiently. Jordan-Squire [Jor15] studies convex optimization over probability measures and implicit differentiation in this context. Bell and Burke [BB08] adapts automatic differentiation to obtain derivatives of implicitly defined functions.

2.4.4 An optimization view of the ReLU, sigmoid, and softmax

In this section we note how the commonly used ReLU, sigmoid, and softmax functions can be interpreted as explicit closed-form solutions to constrained convex optimization (argmin) problems. Bibi, Ghanem, Koltun, and Ranftl [Bib+18] presents another view that interprets other layers as proximal operators and stochastic solvers. We use these as examples to further highlight the power of optimization-based inference, not to provide a new analysis of these layers. The main focus of this thesis is *not* on learning and re-discovering existing activation functions. In this thesis, we rather propose new optimization-based inference layers that do *not* have explicit closed-form solutions like these examples and show that they can still be efficiently turned into differentiable building blocks for end-to-end architectures.

Theorem 2. *The ReLU, defined by* $f(x) = \max\{0, x\}$*, can be interpreted as projecting a point* $x \in \mathbb{R}^n$ *onto the non-negative orthant as*

$$f(x) = \underset{y}{\operatorname{argmin}} \quad \frac{1}{2}||x - y||_2^2 \quad \text{s. t.} \quad y \geq 0. \tag{2.2}$$

Proof. The usual solution can be obtained by looking at the KKT conditions of Equation (2.2). Introducing a dual variable $\lambda \geq 0$ for the inequality constraint, the Lagrangian of Equation (2.2) is

$$L(y, \lambda) = \frac{1}{2}||x - y||_2^2 - \lambda^\top y. \tag{2.3}$$

The stationarity condition $\nabla_y L(y^\star, \lambda^\star) = 0$ gives a way of expressing the primal optimal variable $y^\star$ in terms of the dual optimal variable $\lambda^\star$ as $y^\star = x + \lambda^\star$. Complementary slackness $\lambda_i^\star(x_i + \lambda_i^\star) = 0$ shows that $\lambda_i^\star \in \{0, -x_i\}$. Consider two cases:

- **Case 1:** $x_i \geq 0$. Then $\lambda_i^\star$ must be 0 since we require $\lambda^\star \geq 0$. Thus $y_i^\star = x_i + \lambda_i^\star = x_i$.
- **Case 2:** $x_i < 0$. Then $\lambda_i^\star$ must be $-x_i$ since we require $y \geq 0$. Thus $y_i^\star = x_i + \lambda_i^\star = 0$.

Combining these cases gives the usual solution of $y^\star = \max\{0, x\}$. □

Theorem 3. *The sigmoid or logistic function, defined by* $f(x) = (1 + e^{-x})^{-1}$*, can be interpreted as projecting a point* $x \in \mathbb{R}^n$ *onto the interior of the unit hypercube as*

$$f(x) = \underset{0<y<1}{\operatorname{argmin}} \quad -x^\top y - H_b(y), \tag{2.4}$$

where $H_b(y) = -\left(\sum_i y_i \log y_i + (1 - y_i) \log(1 - y_i)\right)$ *is the binary entropy function.*

Proof. The usual solution can be obtained by looking at the first-order optimality condition of Equation (2.4). The domain of the binary entropy function H_b restricts us to $0 < y < 1$ without needing to explicitly represent this as a constraint in the optimization problem. Let $g(y; x) = -x^\top y - H_b(y)$ be the objective. The first-order optimality condition $\nabla_y g(y^\star; x) = 0$ gives us $-x_i + \log y_i^\star - \log(1 - y_i^\star) = 0$ and thus $y^\star = (1 + e^{-x})^{-1}$. □

Theorem 4. *The softmax, defined by* $f(x)_j = e^{x_j} / \sum_i e^{x_i}$*, can be interpreted as projecting a point* $x \in \mathbb{R}^n$ *onto the interior of the* $(n-1)$*-simplex*

$$\Delta_{n-1} = \{p \in \mathbb{R}^n \mid 1^\top p = 1 \;\text{ and }\; p \geq 0\}$$

as

$$f(x) = \underset{0<y<1}{\operatorname{argmin}} \quad -x^\top y - H(y) \quad \text{s.t.} \quad 1^\top y = 1 \tag{2.5}$$

where $H(y) = -\sum_i y_i \log y_i$ *is the entropy function.*

Proof. The usual solution can be obtained by looking at the KKT conditions of Equation (2.5). Introducing a scalar-valued dual variable ν for the equality constraint, the Lagrangian is

$$L(y, \nu) = -x^\top y - H(y) + \nu(1^\top y - 1) \tag{2.6}$$

The stationarity condition $\nabla_y L(y^\star, \nu^\star) = 0$ gives a way of expressing the primal optimal variable $y^\star$ in terms of the dual optimal variable $\nu^\star$ as

$$y_j^\star = \exp\{x_j - 1 - \nu^\star\}. \tag{2.7}$$

Putting this back into the equality constraint $1^\top y^\star = 1$ gives us $\sum_i \exp\{x_i - 1 - \nu^\star\} = 1$ and thus $\nu^\star = \log \sum_i \exp\{x_i - 1\}$. Substituting this back into Equation (2.7) gives us the usual definition of $y_j = e^{x_j} / \sum_i e^{x_i}$. □

Corollary 1. *A temperature-scaled softmax scales the entropy term in the objective and the sparsemax [MA16] replaces the objective's entropy penalty with a ridge section.*

2.5 Reinforcement Learning and Control

The fields of reinforcement learning (RL) and optimal control typically involve creating agents that act optimally in an environment. These environments can typically be represented as a Markov decision process (MDP) with a continuous or discrete state space and a continuous or discrete action space. The environment often has some oracle-given reward associated with each state and the goal of RL and control is to find a policy that maximizes the cumulative reward achieved.

Using the notation from [Lev17a], *policy search* methods learn a policy $\pi_\theta(u_t|x_t)$ parameterized by θ that predicts a distribution over next action to take given the current state x_t. The goal of policy search is to find a policy that maximizes the expected return

$$\underset{\theta}{\operatorname{argmax}} \ \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \gamma^t r(x_t, u_t)\right], \tag{2.8}$$

where $p_\theta(\tau) = p(x_1)\prod \pi_\theta(u_t|x_t)p(x_{t+1}|x_t, u_t)$ is the distribution over trajectories, $\gamma \in (0, 1]$ is a discount factor, $r(x_t, u_t)$ is the state-action reward at time t, and $p(x_{t+1}|x_t, u_t)$ is the state-transition probability. In many scenarios, the reward r is assumed to be a black-box function that derivative information cannot be obtained from. *Model-free* techniques for policy search typically do not attempt to model the state-transition probability while *model-based* and *control* approaches do.

Control approaches typically provide a policy by planning based on known state transitions. For example, in continuous state-action spaces with deterministic state transitions, the finite-horizon model predictive control problem is

$$\underset{x_{1:T}\in\mathcal{X}, u_{1:T}\in\mathcal{U}}{\operatorname{argmin}} \ \sum_{t=1}^{T} C_t(x_t, u_t) \ \text{ subject to } \ x_{t+1} = f(x_t, u_t), \ \ x_1 = x_{\text{init}}, \tag{2.9}$$

where x_{init} is the current system state, the cost C_t is typically hand-engineered and differentiable, and $x_{t+1} = f(x_t, u_t)$ is the deterministic next-state transition, *i.e.* the point-mass given by $p(x_{t+1}|x_t, u_t)$. While this thesis focuses on the continuous and deterministic setting, control approaches can also be applied in discrete and stochastic settings.

Pure model-free techniques for policy search have demonstrated promising results in many domains by learning *reactive polices* which directly map observations to actions [Mni+13; Oh+16; Gu+16b; Lil+15; Sch+15; Sch+16; Gu+16a]. Despite their success, model-free methods have many drawbacks and limitations, including a lack of interpretability, poor generalization, and a high sample complexity. **Model-based methods** are known to be more sample-efficient than their model-free counterparts. These methods generally rely on learning a dynamics model directly from interactions with the real system and then integrate the learned model into the control policy [Sch97; AQN06; DR11; Hee+15; Boe+14]. More recent approaches use a deep network to learn low-dimensional latent state representations and associated dynamics models in this learned representation. They then apply standard trajectory optimization methods on these learned embeddings [LKS15; Wat+15; Lev+16]. However, these methods still require a manually specified and

hand-tuned cost function, which can become even more difficult in a latent representation. Moreover, there is no guarantee that the learned dynamics model can accurately capture portions of the state space relevant for the task at hand.

To leverage the benefits of both approaches, there has been significant interest in **combining the model-based and model-free paradigms.** In particular, much attention has been dedicated to utilizing model-based priors to accelerate the model-free learning process. For instance, synthetic training data can be generated by model-based control algorithms to guide the policy search or prime a model-free policy [Sut90; TBS10; LA14; Gu+16b; Ven+16; Lev+16; Che+17a; Nag+17; Sun+17]. [Ban+17] learns a controller and then distills it to a neural network policy which is then fine-tuned with model-free policy learning. However, this line of work usually keeps the model separate from the learned policy.

Alternatively, the policy can include an **explicit planning module** which *leverages learned models* of the system or environment, both of which are learned through model-free techniques. For example, the classic Dyna-Q algorithm [Sut90] simultaneously learns a model of the environment and uses it to plan. More recent work has explored incorporating such structure into deep networks and learning the policies in an end-to-end fashion. Tamar, Wu, Thomas, Levine, and Abbeel [Tam+16] uses a recurrent network to predict the value function by approximating the value iteration algorithm with convolutional layers. Karkus, Hsu, and Lee [KHL17] connects a dynamics model to a planning algorithm and formulates the policy as a structured recurrent network. Silver, Hasselt, Hessel, Schaul, Guez, Harley, Dulac-Arnold, Reichert, Rabinowitz, Barreto, et al. [Sil+16] and Oh, Singh, and Lee [OSL17] perform multiple rollouts using an abstract dynamics model to predict the value function. A similar approach is taken by Weber, Racanière, Reichert, Buesing, Guez, Rezende, Badia, Vinyals, Heess, Li, et al. [Web+17] but directly predicts the next action and reward from rollouts of an explicit environment model. Farquhar, Rocktäschel, Igl, and Whiteson [Far+17] extends model-free approaches, such as DQN [Mni+15] and A3C [Mni+16], by planning with a tree-structured neural network to predict the cost-to-go. While these approaches have demonstrated impressive results in discrete state and action spaces, they are not applicable to continuous control problems.

To tackle continuous state and action spaces, Pascanu, Li, Vinyals, Heess, Buesing, Racanière, Reichert, Weber, Wierstra, and Battaglia [Pas+17a] propose a neural architecture which uses an abstract environmental model to plan and is trained directly from an external task loss. Pong, Gu, Dalal, and Levine [Pon+18] learn goal-conditioned value functions and use them to plan single or multiple steps of actions in an MPC fashion. Similarly, Pathak, Mahmoudieh, Luo, Agrawal, Chen, Shentu, Shelhamer, Malik, Efros, and Darrell [Pat+18] train a goal-conditioned policy to perform rollouts in an abstract feature space but ground the policy with a loss term which corresponds to true dynamics data. The aforementioned approaches can be interpreted as a distilled optimal controller which does not separate components for the cost and dynamics. Taking this analogy further, another strategy is to differentiate through an optimal control algorithm itself. Okada, Rigazio, and Aoshima [ORA17] and Pereira, Fan, An, and Theodorou [Per+18] present a way to differentiate through path integral optimal control [Wil+16; WAT17] and learn a planning policy end-to-end. Srinivas, Jabri, Abbeel, Levine, and Finn [Sri+18] shows how to embed

differentiable planning (unrolled gradient descent over actions) within a goal-directed policy. In a similar vein, Tamar, Thomas, Zhang, Levine, and Abbeel [Tam+17] differentiates through an iterative LQR (iLQR) solver [LT04; XLH17; TMT14] to learn a cost-shaping term offline. This shaping term enables a shorter horizon controller to approximate the behavior of a solver with a longer horizon to save computation during runtime.

Part I

Foundations

Chapter 3

OptNet: Differentiable Optimization as a Layer in Deep Learning

This chapter describes OptNet, a network architecture that integrates optimization problems (here, specifically in the form of quadratic programs) as individual layers in larger end-to-end trainable deep networks. These layers encode constraints and complex dependencies between the hidden states that traditional convolutional and fully-connected layers often cannot capture. We explore the foundations for such an architecture: we show how techniques from sensitivity analysis, bilevel optimization, and implicit differentiation can be used to exactly differentiate through these layers and with respect to layer parameters; we develop a highly efficient solver for these layers that exploits fast GPU-based batch solves within a primal-dual interior point method, and which provides backpropagation gradients with virtually no additional cost on top of the solve; and we highlight the application of these approaches in several problems. In one notable example, we show that the method is capable of learning to play mini-Sudoku (4x4) given just input and output games, with no a priori information about the rules of the game; this highlights the ability of our architecture to learn hard constraints better than other neural architectures.

The contents of this chapter have been previously published at ICML 2017 in Amos and Kolter [AK17].

3.1 Introduction

In this chapter, we consider how to treat exact, constrained optimization as an individual layer within a deep learning architecture. Unlike traditional feedforward networks, where the output of each layer is a relatively simple (though non-linear) function of the previous layer, our optimization framework allows for individual layers to capture much richer behavior, expressing complex operations that in total can reduce the overall depth of the network while preserving richness of representation. Specifically, we build a framework where the output of the $i+1$th layer in a network is the *solution* to a constrained optimization problem based upon previous layers. This framework naturally encompasses

a wide variety of inference problems expressed within a neural network, allowing for the potential of much richer end-to-end training for complex tasks that require such inference procedures.

Concretely, in this chapter we specifically consider the task of solving small quadratic programs as individual layers. These optimization problems are well-suited to capturing interesting behavior and can be efficiently solved with GPUs. Specifically, we consider layers of the form

$$\begin{aligned} z_{i+1} = \underset{z}{\text{argmin}} \;\; & \frac{1}{2} z^T Q(z_i) z + q(z_i)^T z \\ \text{subject to} \;\; & A(z_i) z = b(z_i) \\ & G(z_i) z \leq h(z_i) \end{aligned} \tag{3.1}$$

where z is the optimization variable, $Q(z_i)$, $q(z_i)$, $A(z_i)$, $b(z_i)$, $G(z_i)$, and $h(z_i)$ are parameters of the optimization problem. As the notation suggests, these parameters can depend in any differentiable way on the previous layer z_i, and which can eventually be optimized just like any other weights in a neural network. These layers can be learned by taking the gradients of some loss function with respect to the parameters. In this chapter, we derive the gradients of (3.1) by taking matrix differentials of the KKT conditions of the optimization problem at its solution.

In order to the make the approach practical for larger networks, we develop a custom solver which can simultaneously solve multiple small QPs in batch form. We do so by developing a custom primal-dual interior point method tailored specifically to dense batch operations on a GPU. In total, the solver can solve batches of quadratic programs over 100 times faster than existing highly tuned quadratic programming solvers such as Gurobi and CPLEX. One crucial algorithmic insight in the solver is that by using a specific factorization of the primal-dual interior point update, we can obtain a backward pass over the optimization layer virtually "for free" (i.e., requiring no additional factorization once the optimization problem itself has been solved). Together, these innovations enable parameterized optimization problems to be inserted within the architecture of existing deep networks.

We begin by highlighting background and related work, and then present our optimization layer itself. Using matrix differentials we derive rules for computing all the necessary backpropagation updates. We then detail our specific solver for these quadratic programs, based upon a state-of-the-art primal-dual interior point method, and highlight the novel elements as they apply to our formulation, such as the aforementioned fact that we can compute backpropagation at very little additional cost. We then provide experimental results that demonstrate the capabilities of the architecture, highlighting potential tasks that these architectures can solve, and illustrating improvements upon existing approaches.

3.2 Connections to related work

Optimization plays a key role in modeling complex phenomena and providing concrete decision making processes in sophisticated environments. A full treatment of optimiza-

tion applications is beyond our scope [BV04] but these methods have bound applicability in control frameworks [ML99; SB11]; numerous statistical and mathematical formalisms [SNW12], and physical simulation problems like rigid body dynamics [Löt84].

We contrast the OptNet approach to the related differentiable optimization-based inference methods in Section 2.4. We do **not** use analytical differentiation or unrolling, as the optimization problem we consider is constrained and does not have an explicit closed-form solution. The argmin differentiation work we discuss in Section 2.4.3 is the most closely related to this chapter. Johnson, Duvenaud, Wiltschko, Adams, and Datta [Joh+16] performs implicit differentiation on (multi-)convex objectives with coordinate subspace constraints, but don't consider inequality constraints and don't consider in detail general linear equality constraints. Their optimization problem is only in the final layer of a variational inference network while we propose to insert optimization problems anywhere in the network. Therefore a special case of OptNet layers (with no inequality constraints) has a natural interpretation in terms of Gaussian inference, and so Gaussian graphical models (or CRF ideas more generally) provide tools for making the computation more efficient and interpreting or constraining its structure. Similarly, the older work of Mairal, Bach, and Ponce [MBP12] considered argmin differentiation for a LASSO problem, deriving specific rules for this case, and presenting an efficient algorithm based upon our ability to solve the LASSO problem efficiently.

In this chapter, we use implicit differentiation [DR09; GW08] and techniques from matrix differential calculus [MN88] to derive the gradients from the KKT matrix of the problem we are interested in. A notable difference from other work within ML that we are aware of, is that we analytically differentiate through inequality as well as just equality constraints, but differentiating the complementarity conditions; this differs from e.g., Gould, Fernando, Cherian, Anderson, Santa Cruz, and Guo [Gou+16] where they instead approximately convert the problem to an unconstrained one via a barrier method. We have also developed methods to make this approach practical and reasonably scalable within the context of deep architectures.

3.3 Solving optimization within a neural network

Although in the most general form, an OptNet layer can be any optimization problem, in this chapter we will study OptNet layers defined by a quadratic program

$$\begin{array}{ll} \underset{z}{\text{minimize}} & \frac{1}{2} z^T Q z + q^T z \\ \text{subject to} & Az = b, \; Gz \leq h \end{array} \tag{3.2}$$

where $z \in \mathbb{R}^n$ is our optimization variable $Q \in \mathbb{R}^{n \times n} \succeq 0$ (a positive semidefinite matrix), $q \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $G \in \mathbb{R}^{p \times n}$ and $h \in \mathbb{R}^p$ are problem data, and leaving out the dependence on the previous layer z_i as we showed in (3.1) for notational convenience. As is well-known, these problems can be solved in polynomial time using a variety of methods; if one desires exact (to numerical precision) solutions to these problems, then primal-dual interior point methods, as we will use in a later section, are the current state of the art in

solution methods. In the neural network setting, the *optimal solution* (or more generally, a *subset of the optimal solution*) of this optimization problems becomes the output of our layer, denoted z_{i+1}, and any of the problem data Q, q, A, b, G, h can depend on the value of the previous layer z_i. The forward pass in our OptNet architecture thus involves simply setting up and finding the solution to this optimization problem.

Training deep architectures, however, requires that we not just have a forward pass in our network but also a backward pass. This requires that we compute the derivative of the solution to the QP with respect to its input parameters, a general topic we topic we discussed previously. To obtain these derivatives, we differentiate the KKT conditions (sufficient and necessary conditions for optimality) of (3.2) at a solution to the problem using techniques from matrix differential calculus [MN88]. Our analysis here can be extended to more general convex optimization problems.

The Lagrangian of (3.2) is given by

$$L(z, \nu, \lambda) = \frac{1}{2} z^T Q z + q^T z + \nu^T (Az - b) + \lambda^T (Gz - h) \tag{3.3}$$

where ν are the dual variables on the equality constraints and $\lambda \geq 0$ are the dual variables on the inequality constraints. The KKT conditions for stationarity, primal feasibility, and complementary slackness are

$$\begin{aligned} Qz^\star + q + A^T \nu^\star + G^T \lambda^\star &= 0 \\ Az^\star - b &= 0 \\ D(\lambda^\star)(Gz^\star - h) &= 0, \end{aligned} \tag{3.4}$$

where $D(\cdot)$ creates a diagonal matrix from a vector and $z^\star$, $\nu^\star$ and $\lambda^\star$ are the optimal primal and dual variables. Taking the differentials of these conditions gives the equations

$$\begin{aligned} \mathsf{d}Qz^\star + Q\mathsf{d}z + \mathsf{d}q + \mathsf{d}A^T \nu^\star + \\ A^T \mathsf{d}\nu + \mathsf{d}G^T \lambda^\star + G^T \mathsf{d}\lambda &= 0 \\ \mathsf{d}Az^\star + A\mathsf{d}z - \mathsf{d}b &= 0 \\ D(Gz^\star - h)\mathsf{d}\lambda + D(\lambda^\star)(\mathsf{d}Gz^\star + G\mathsf{d}z - \mathsf{d}h) &= 0 \end{aligned} \tag{3.5}$$

or written more compactly in matrix form

$$\begin{bmatrix} Q & G^T & A^T \\ D(\lambda^\star)G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathsf{d}z \\ \mathsf{d}\lambda \\ \mathsf{d}\nu \end{bmatrix} = \begin{bmatrix} -\mathsf{d}Qz^\star - \mathsf{d}q - \mathsf{d}G^T \lambda^\star - \mathsf{d}A^T \nu^\star \\ -D(\lambda^\star)\mathsf{d}Gz^\star + D(\lambda^\star)\mathsf{d}h \\ -\mathsf{d}Az^\star + \mathsf{d}b \end{bmatrix}. \tag{3.6}$$

Using these equations, we can form the Jacobians of $z^\star$ (or $\lambda^\star$ and $\nu^\star$, though we don't consider this case here), with respect to any of the data parameters. For example, if we wished to compute the Jacobian $\frac{\partial z^\star}{\partial b} \in \mathbb{R}^{n \times m}$, we would simply substitute $\mathsf{d}b = I$ (and set all other differential terms in the right hand side to zero), solve the equation, and the resulting value of $\mathsf{d}z$ would be the desired Jacobian.

In the backpropagation algorithm, however, we never want to explicitly form the actual Jacobian matrices, but rather want to form the left matrix-vector product with some

previous backward pass vector $\frac{\partial \ell}{\partial z^\star} \in \mathbb{R}^n$, i.e., $\frac{\partial \ell}{\partial z^\star}\frac{\partial z^\star}{\partial b}$. We can do this efficiently by noting the solution for the $(\mathtt{d}z, \mathtt{d}\lambda, \mathtt{d}\nu)$ involves multiplying the *inverse* of the left-hand-side matrix in (3.6) by some right hand side. Thus, if we multiply the backward pass vector by the transpose of the differential matrix

$$\begin{bmatrix} d_z \\ d_\lambda \\ d_\nu \end{bmatrix} = - \begin{bmatrix} Q & G^T D(\lambda^\star) & A^T \\ G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \nabla_{z^\star}\ell \\ 0 \\ 0 \end{bmatrix} \tag{3.7}$$

then the relevant gradients with respect to all the QP parameters can be given by

$$\begin{aligned} \nabla_Q \ell &= \frac{1}{2}(d_z z^T + z d_z^T) & \nabla_q \ell &= d_z \\ \nabla_A \ell &= d_\nu z^T + \nu d_z^T & \nabla_b \ell &= -d_\nu \\ \nabla_G \ell &= D(\lambda^\star)(d_\lambda z^T + \lambda d_z^T) & \nabla_h \ell &= -D(\lambda^\star) d_\lambda \end{aligned} \tag{3.8}$$

where as in standard backpropagation, all these terms are at most the size of the parameter matrices. We note that some of these parameters should depend on the previous layer z_i and the gradients with respect to the previous layer can be obtained through the chain rule. As we will see in the next section, the solution to an interior point method in fact already provides a factorization we can use to compute these gradient efficiently.

3.3.1 An efficient batched QP solver

Deep networks are typically trained in mini-batches to take advantage of efficient data-parallel GPU operations. Without mini-batching on the GPU, many modern deep learning architectures become intractable for all practical purposes. However, today's state-of-the-art QP solvers like Gurobi and CPLEX do not have the capability of solving multiple optimization problems on the GPU in parallel across the entire minibatch. This makes larger OptNet layers become quickly intractable compared to a fully-connected layer with the same number of parameters.

To overcome this performance bottleneck in our quadratic program layers, we have implemented a GPU-based primal-dual interior point method (PDIPM) based on Mattingley and Boyd [MB12] that solves a batch of quadratic programs, and which provides the necessary gradients needed to train these in an end-to-end fashion. Our performance experiments in Section 3.4.1 shows that our solver is significantly faster than the standard non-batch solvers Gurobi and CPLEX.

Following the method of Mattingley and Boyd [MB12], our solver introduces slack variables on the inequality constraints and iteratively minimizes the residuals from the KKT conditions over the primal variable $z \in \mathbb{R}^n$, slack variable $s \in \mathbb{R}^p$, and dual variables $\nu \in \mathbb{R}^m$ associated with the equality constraints and $\lambda \in \mathbb{R}^p$ associated with the inequality constraints. Each iteration computes the affine scaling directions by solving

$$K \begin{bmatrix} \Delta z^{\text{aff}} \\ \Delta s^{\text{aff}} \\ \Delta \lambda^{\text{aff}} \\ \Delta \nu^{\text{aff}} \end{bmatrix} = \begin{bmatrix} -(A^T\nu + G^T\lambda + Qz + q) \\ -S\lambda \\ -(Gz + s - h) \\ -(Az - b) \end{bmatrix} \tag{3.9}$$

where

$$K = \begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & D(\lambda) & D(s) & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix},$$

then centering-plus-corrector directions by solving

$$K \begin{bmatrix} \Delta z^{\text{cc}} \\ \Delta s^{\text{cc}} \\ \Delta \lambda^{\text{cc}} \\ \Delta \nu^{\text{cc}} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma\mu 1 - D(\Delta s^{\text{aff}})\Delta\lambda^{\text{aff}} \\ 0 \\ 0 \end{bmatrix}, \tag{3.10}$$

where $\mu = s^T\lambda/p$ is the duality gap and σ is defined in Mattingley and Boyd [MB12]. Each variable v is updated with $\Delta v = \Delta v^{\text{aff}} + \Delta v^{\text{cc}}$ using an appropriate step size. We actually solve a symmetrized version of the KKT conditions, obtained by scaling the second row block by $D(1/s)$. We analytically decompose these systems into smaller symmetric systems and pre-factorize portions of them that don't change (i.e. that don't involve $D(\lambda/s)$ between iterations). We have implemented a batched version of this method with the PyTorch library [Pas+17b] and have released it as an open source library at `https://github.com/locuslab/qpth`. It uses a custom CUBLAS extension that provides an interface to solve multiple matrix factorizations and solves in parallel, and which provides the necessary backpropagation gradients for their use in an end-to-end learning system.

Efficiently computing gradients

A key point of the particular form of primal-dual interior point method that we employ is that it is possible to compute the backward pass gradients "for free" after solving the original QP, without an additional matrix factorization or solve. Specifically, at each iteration in the primal-dual interior point, we are computing an LU decomposition of the matrix K_{sym}.[1] This matrix is essentially a symmetrized version of the matrix needed for computing the backpropagated gradients, and we can similarly compute the $d_{z,\lambda,\nu}$ terms by solving the linear system

$$K_{\text{sym}} \begin{bmatrix} d_z \\ d_s \\ \tilde{d}_\lambda \\ d_\nu \end{bmatrix} = \begin{bmatrix} \nabla_{z_{i+1}}\ell \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{3.11}$$

where $\tilde{d}_\lambda = D(\lambda^\star)d_\lambda$ for d_λ as defined in (3.7). Thus, all the backward pass gradients can be computed using the factored KKT matrix at the solution. Crucially, because the bottleneck

[1] We actually perform an LU decomposition of a certain subset of the matrix formed by eliminating variables to create only a $p \times p$ matrix (the number of inequality constraints) that needs to be factor during each iteration of the primal-dual algorithm, and one $m \times m$ and one $n \times n$ matrix once at the start of the primal-dual algorithm, though we omit the detail here. We also use an LU decomposition as this routine is provided in batch form by CUBLAS, but could potentially use a (faster) Cholesky factorization if and when the appropriate functionality is added to CUBLAS).

of solving this linear system is computing the factorization of the KKT matrix (cubic time as opposed to the quadratic time for solving via backsubstitution once the factorization is computed), the additional time requirements for computing all the necessary gradients in the backward pass is virtually nonexistent compared with the time of computing the solution. To the best of our knowledge, this is the first time that this fact has been exploited in the context of learning end-to-end systems.

3.3.2 Properties and representational power

In this section we briefly highlight some of the mathematical properties of OptNet layers. The proofs here are straightforward and are mostly based upon well-known results in convex analysis. The first result simply highlights that (because the solution of strictly convex QPs is continuous), that OptNet layers are subdifferentiable everywhere, and differentiable at all but a measure-zero set of points.

Theorem 5. *Let $z^\star(\theta)$ be the output of an OptNet layer, where $\theta = \{Q, p, A, b, G, h\}$. Assuming $Q \succ 0$ and that A has full row rank, then $z^\star(\theta)$ is subdifferentiable everywhere: $\partial z^\star(\theta) \neq \varnothing$, where $\partial z^\star(\theta)$ denotes the Clarke generalized subdifferential [Cla75] (an extension of the subgradient to non-convex functions), and has a single unique element (the Jacobian) for all but a measure zero set of points θ.*

Proof. The fact that an OptNet layer is subdifferentiable from strictly convex QPs ($Q \succ 0$) follows directly from the well-known result that the solution of a strictly convex QP is continuous (though not everywhere differentiable). Our proof essentially just boils down to showing this fact, though we do so by explicitly showing that there *is* a unique solution to the Jacobian equations (3.6) that we presented earlier, except on a measure zero set. This measure zero set consists of QPs with degenerate solutions, points where inequality constraints can hold with equality yet also have zero-valued dual variables. For simplicity we assume that A has full row rank, but this can be relaxed.

From the complementarity condition, we have that at a primal dual solution $(z^\star, \lambda^\star, \nu^\star)$

$$\begin{aligned} (Gz^\star - h)_i < 0 &\rightarrow \lambda_i^\star = 0 \\ \lambda_i^\star > 0 &\rightarrow (Gz^\star - h)_i = 0 \end{aligned} \tag{3.12}$$

(i.e., we cannot have both these terms non-zero).

First we consider the (typical) case where exactly one of $(Gz^\star - h)_i$ and $\lambda_i^\star$ is zero. Then the KKT differential matrix

$$\begin{bmatrix} Q & G^T & A^T \\ D(\lambda^\star)G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix} \tag{3.13}$$

(the left hand side of (3.6)) is non-singular. To see this, note that if we let $\mathcal{I}$ be the set where $\lambda_i^\star > 0$, then the matrix

$$\begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & D(Gz^\star - h)_{\mathcal{I}} & 0 \\ A & 0 & 0 \end{bmatrix} = \begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & 0 & 0 \\ A & 0 & 0 \end{bmatrix} \tag{3.14}$$

is non-singular (scaling the second block by $D(\lambda^\star)^{-1}$ gives a standard KKT system [BV04, Section 10.4], which is nonsingular for invertible Q and $[G_{\mathcal{I}}^T \quad A^T]$ with full column rank, which must hold due to our condition on A and the fact that there must be less than n total tight constraints at the solution. Also note that for any $i \notin \mathcal{I}$, only the $D(Gz^\star - h)_{ii}$ term is non-zero for the entire row in the second block of the matrix. Thus, if we want to solve the system

$$\begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & D(Gz^\star - h)_{\mathcal{I}} & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda \\ \nu \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{3.15}$$

we simply first set $\lambda_i = b_i/(Gz^\star - h)_i$ for $i \notin \mathcal{I}$ and then solve the nonsingular system

$$\begin{bmatrix} Q & G_{\mathcal{I}}^T & A^T \\ D(\lambda^\star)G_{\mathcal{I}} & 0 & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda_{\mathcal{I}} \\ \nu \end{bmatrix} = \begin{bmatrix} a - G_{\bar{\mathcal{I}}}^T \lambda_{\bar{\mathcal{I}}} \\ b_{\mathcal{I}} \\ c. \end{bmatrix} \tag{3.16}$$

Alternatively, suppose that we have both $\lambda_i^\star = 0$ and $(Gz^\star - h)_i = 0$. Then although the KKT matrix is now singular (any row for which $\lambda_i^\star = 0$ and $(Gz^\star - h)_i = 0$ will be all zero), there still exists a solution to the system (3.6), because the right hand side is always in the range of $D(\lambda^\star)$ and so will also be zero for these rows. In this case there will no longer be a *unique* solution, corresponding to the subdifferentiable but not differentiable case. □

The next two results show the representational power of the OptNet layer, specifically how an OptNet layer compares to the common linear layer followed by a ReLU activation. The first theorem shows that an OptNet layer can approximate arbitrary elementwise piecewise-linear functions, and so among other things can represent a ReLU layer.

Theorem 6. *Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be an elementwise piecewise linear function with k linear regions. Then the function can be represented as an OptNet layer using $O(nk)$ parameters. Additionally, the layer $z_{i+1} = \max\{Wz_i + b, 0\}$ for $W \in \mathbb{R}^{n \times m}, b \in \mathbb{R}^n$ can be represented by an OptNet layer with $O(mn)$ parameters.*

Proof. The proof that an OptNet layer can represent any piecewise linear univariate function relies on the fact that we can represent any such function in "sum-of-max" form

$$f(x) = \sum_{i=1}^{k} w_i \max\{a_i x + b, 0\} \tag{3.17}$$

where $w_i \in \{-1, 1\}$, $a_i, b_i \in \mathbb{R}$ (to do so, simply proceed left to right along the breakpoints of the function adding a properly scaled linear term to fit the next piecewise section). The OptNet layer simply represents this function directly.

That is, we encode the optimization problem

$$\begin{aligned} \underset{z \in \mathbb{R}, t \in \mathbb{R}^k}{\text{minimize}} \quad & \|t\|_2^2 + (z - w^T t)^2 \\ \text{subject to} \quad & a_i x + b_i \leq t_i, \quad i = 1, \ldots, k \end{aligned} \tag{3.18}$$

Clearly, the objective here is minimized when $z = w^T t$, and t is as small as possible, meaning each t must either be at its bound $a_i x + b \leq t_i$ or, if $a_i x + b < 0$, then $t_i = 0$ will be the optimal solution due to the objective function. To obtain a multivariate but elementwise function, we simply apply this function to each coordinate of the input x.

To see the specific case of a ReLU network, note that the layer

$$z = \max\{Wx + b, 0\} \tag{3.19}$$

is simply equivalent to the OptNet problem

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \|z - Wx - b\|_2^2 \\ \text{subject to} \quad & z \geq 0. \end{aligned} \tag{3.20}$$

□

Finally, we show that the converse does not hold: that there are function representable by an OptNet layer which cannot be represented exactly by a two-layer ReLU layer, which take exponentially many units to approximate (known to be a universal function approximator). A simple example of such a layer (and one which we use in the proof) is just the max over three linear functions $f(z) = \max\{a_1^T x, a_2^T x, a_3^T x\}$.

Theorem 7. *Let $f(z) : \mathbb{R}^n \to \mathbb{R}$ be a scalar-valued function specified by an OptNet layer with p parameters. Conversely, let $f'(z) = \sum_{i=1}^m w_i \max\{a_i^T z + b_i, 0\}$ be the output of a two-layer ReLU network. Then there exist functions that the ReLU network cannot represent exactly over all of $\mathbb{R}$, and which require $O(c^p)$ parameters to approximate over a finite region.*

Proof. The final theorem simply states that a two-layer ReLU network (more specifically, a ReLU followed by a linear layer, which is sufficient to achieve a universal function approximator), can often require exponentially many more units to approximate a function specified by an OptNet layer. That is, we consider a single-output ReLU network, much like in the previous section, but defined for multi-variate inputs.

$$f(x) = \sum_{i=1}^m w_i \max\{a_i^T x + b, 0\} \tag{3.21}$$

Although there are many functions that such a network cannot represent, for illustration we consider a simple case of a maximum of three linear functions

$$f'(x) = \max\{a_1^T x, a_2^T x, a_3^T x\} \tag{3.22}$$

To see why a ReLU is not capable of representing this function exactly, even for $x \in \mathbb{R}^2$, note that any sum-of-max function, due to the nature of the term $\max\{a_i^T x + b_i, 0\}$ as stated above must have "creases" (breakpoints in the piecewise linear function), than span the entire input space; this is in contrast to the max terms, which can have creases that only partially span the space. This is illustrated in Figure 3.1. It is apparent, therefore, that the two-layer ReLU cannot exactly approximate the three maximum term (any ReLU

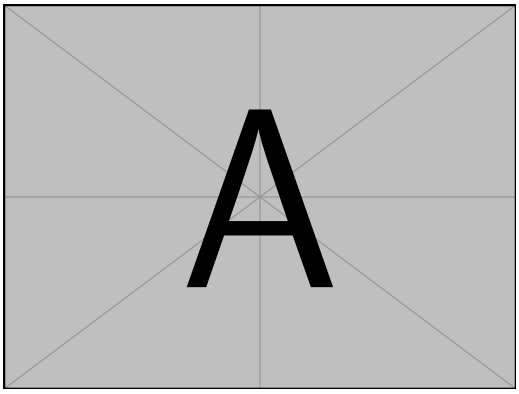


Figure 3.1: Creases for a three-term pointwise maximum (left), and a ReLU network (right).

network would necessarily have a crease going through one of the linear region of the original function). Yet this max function can be captured by a simple OptNet layer

$$\begin{aligned} &\underset{z}{\text{minimize}} \;\; z^2 \\ &\text{subject to} \;\; a_i^T x \leq z, \;\; i = 1, \ldots, 3. \end{aligned} \tag{3.23}$$

The fact that the ReLU network is a universal function approximator means that the we *are* able to approximate the three-max term, but to do so means that we require a dense covering of points over the input space, choose an equal number of ReLU terms, then choose coefficients such that we approximate the underlying function on this points; however, for a large enough radius this will require an exponential size covering to approximate the underlying function arbitrarily closely. □

Although the example here in this proof is quite simple (and perhaps somewhat limited, since for example the function can be exactly approximated using a "Maxout" network), there are a number of other such functions for which we have been unable to find any compact representation. For example, projection of a point on to the simplex is easily written as the OptNet layer

$$\begin{aligned} &\underset{z}{\text{minimize}} \;\; \|z - x\|_2^2 \\ &\text{subject to} \;\; z \geq 0, 1^T z = 1 \end{aligned} \tag{3.24}$$

yet it does not seem possible to represent this in closed form as a simple network: the closed form solution of such a projection operator requires sorting or finding a particular median term of the data [Duc+08], which is not feasible with a single layer for any form of network that we are aware of. Yet for simplicity we stated the theorem above using just ReLU networks and a straightforward example that works even in two dimensions.

3.3.3 Limitations of the method

Although, as we will show shortly, the OptNet layer has several strong points, we also want to highlight the potential drawbacks of this approach. First, although, with an

efficient batch solver, integrating an OptNet layer into existing deep learning architectures is potentially practical, we do note that solving optimization problems exactly as we do here has has cubic complexity in the number of variables and/or constraints. This contrasts with the quadratic complexity of standard feedforward layers. This means that we *are* ultimately limited to settings where the number of hidden variables in an OptNet layer is not too large (less than 1000 dimensions seems to be the limits of what we currently find to the be practical, and substantially less if one wants real-time results for an architecture).

Secondly, there are many improvements to the OptNet layers that are still possible. Our QP solver, for instance, uses fully dense matrix operations, which makes the solves very efficient for GPU solutions, and which also makes sense for our general setting where the coefficients of the quadratic problem can be learned. However, for setting many real-world optimization problems (and hence for architectures that wish to more closely mimic some real-world optimization problem), there is often substantial structure (e.g., sparsity), in the data matrices that can be exploited for efficiency. There is of course no prohibition of incorporating sparse matrix methods into the fast custom solver, but doing so would require substantial added complexity, especially regarding efforts like finding minimum fill orderings for different sparsity patterns of the KKT systems. In our open source solver `qpth`, we have started experimenting with cuSOLVER's batched sparse QR factorizations and solves.

Lastly, we note that while the OptNet layers can be trained just as any neural network layer, since they are a new creation and since they have manifolds in the parameter space which have no effect on the resulting solution (e.g., scaling the rows of a constraint matrix and its right hand side does not change the optimization problem), there is admittedly more tuning required to get these to work. This situation is common when developing new neural network architectures and has also been reported in the similar architecture of Schmidt and Roth [SR14]. Our hope is that techniques for overcoming some of the challenges in learning these layers will continue to be developed in future work.

3.4 Experimental results

In this section, we present several experimental results that highlight the capabilities of the QP OptNet layer. Specifically we look at 1) computational efficiency over exiting solvers; 2) the ability to improve upon existing convex problems such as those used in signal denoising; 3) integrating the architecture into an generic deep learning architectures; and 4) performance of our approach on a problem that is challenging for current approaches. In particular, we want to emphasize the results of our system on learning the game of (4x4) mini-Sudoku, a well-known logical puzzle; our layer is able to directly learn the necessary constraints using just gradient information and no a priori knowledge of the rules of Sudoku. The code and data for our experiments are open sourced in the `icml2017` branch of `https://github.com/locuslab/optnet` and our batched QP solver is available as a library at `https://github.com/locuslab/qpth`.

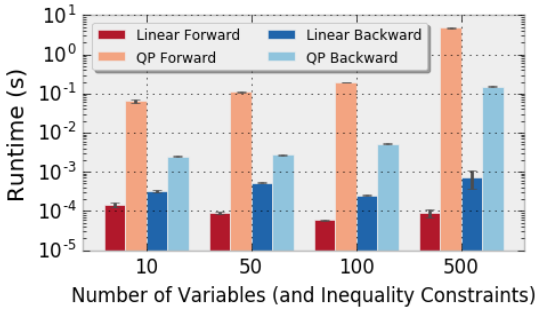


Figure 3.2: Performance of a linear layer and a QP layer. (Batch size 128)

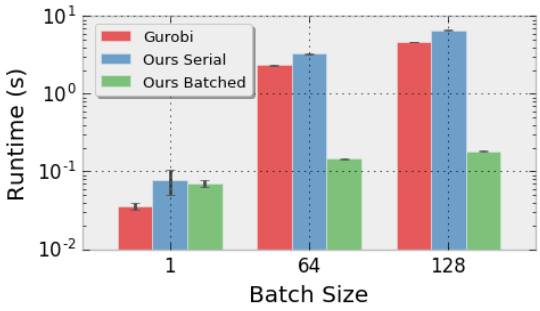


Figure 3.3: Performance of Gurobi and our QP solver.

3.4.1 Batch QP solver performance

All of the OptNet performance results in this section are run on an unloaded Titan X GPU. Gurobi is run on an unloaded quad-core Intel Core i7-5960X CPU @ 3.00GHz.

Our OptNet layers are much more computationally expensive than a linear or convolutional layer and a natural question is to ask what the performance difference is. We set up an experiment comparing a linear layer to a QP OptNet layer with a mini-batch size of 128 on CUDA with randomly generated input vectors sized 10, 50, 100, and 500. Each layer maps this input to an output of the same dimension; the linear layer does this with a batched matrix-vector multiplication and the OptNet layer does this by taking the argmin of a random QP that has the same number of inequality constraints as the dimensionality of the problem. Figure 3.2 shows the profiling results (averaged over 10 trials) of the forward and backward passes. The OptNet layer is significantly slower than the linear layer as expected, yet still tractable in many practical contexts.

Our next experiment illustrates why standard baseline QP solvers like CPLEX and Gurobi without batch support are too computationally expensive for QP OptNet layers to be tractable. We set up random QP of the form (3.1) that have 100 variables and 100 inequality constraints in Gurobi and in the serialized and batched versions of our solver `qpth` and vary the batch size.[2]

Figure 3.3 shows the means and standard deviations of running each trial 10 times, showing that our batched solver outperforms Gurobi, itself a highly tuned solver for reasonable batch sizes. For the minibatch size of 128, we solve all problems in an average of 0.18 seconds, whereas Gurobi tasks an average of 4.7 seconds. In the context of training a deep architecture this type of speed difference for a single minibatch can make the difference between a practical and a completely unusable solution.

[2]Experimental details: we sample entries of a matrix U from a random uniform distribution and set $Q = U^T U + 10^{-3} I$, sample G with random normal entries, and set h by selecting generating some z_0 random normal and s_0 random uniform and setting $h = Gz_0 + s_0$ (we didn't include equality constraints just for simplicity, and since the number of inequality constraints in the primary driver of complexity for the iterations in a primal-dual interior point method). The choice of h guarantees the problem is feasible.

3.4.2 Total variation denoising

Our next experiment studies how we can use the OptNet architecture to *improve* upon signal processing techniques that currently use convex optimization as a basis. Specifically, our goal in this case is to denoise a noisy 1D signal given training data consistency of noisy and clean signals generated from the same distribution. Such problems are often addressed by convex optimization procedures, and (1D) total variation denoising is a particularly common and simple approach. Specifically, the total variation denoising approach attempts to smooth some noisy observed signal y by solving the optimization problem

$$\underset{z}{\text{argmin}} \frac{1}{2}||y - z|| + \lambda||Dz||_1 \tag{3.25}$$

where D is the first-order differencing operation, which can be expressed in matrix form by a matrix with rows $D_i = e_i - e_{i+1}$ Penalizing the ℓ_1 norm of the signal *difference* encourages this difference to be sparse, i.e., the number of changepoints of the signal is small, and we end up approximating y by a (roughly) piecewise constant function.

To test this approach and competing ones on a denoising task, we generate piecewise constant signals (which are the desired outputs of the learning algorithm) and corrupt them with independent Gaussian noise (which form the inputs to the learning algorithm). Table 3.1 shows the error rate of these four approaches.

Baseline: Total variation denoising

To establish a baseline for denoising performance with total variation, we run the above optimization problem varying values of λ between 0 and 100. The procedure performs best with a choice of $\lambda \approx 13$, and achieves a minimum test MSE on our task of about 16.5 (the units here are unimportant, the only relevant quantity is the relative performances of the different algorithms).

Baseline: Learning with a fully-connected neural network

An alternative approach to denoising is by learning from data. A function $f_\theta(x)$ parameterized by θ can be used to predict the original signal. The optimal θ can be learned by using the mean squared error between the true and predicted signals. Denoising is typically a difficult function to learn and Table 3.1 shows that a fully-connected neural network perform substantially worse on this denoising task than the convex optimization problem. Figure 3.4 shows the error of the fully connected network on the denoising task.

Learning the differencing operator with OptNet

Between the feedforward neural network approach and the convex total variation optimization, we could instead use a generic OptNet layers that effectively allowed us to solve (3.25) using *any* denoising matrix, which we randomly initialize. While the accuracy here is substantially lower than even the fully connected case, this is largely the result of learning an over-regularized solution to D. This is indeed a point that should be addressed in future

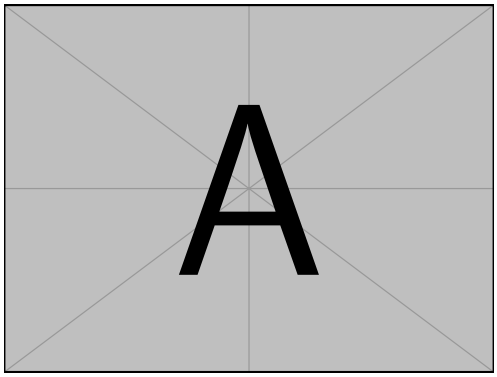


Figure 3.4: Error of the fully connected network for denoising

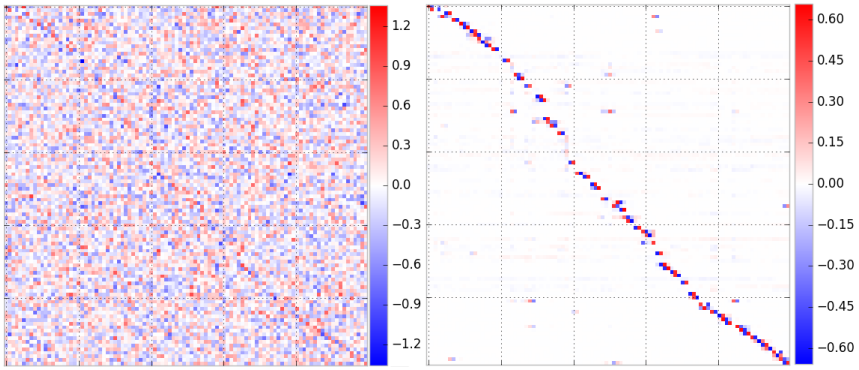


Figure 3.5: Initial and learned difference operators for denoising.

work (we refer back to our comments in the previous section on the potential challenges of training these layers), but the point we want to highlight here is that the OptNet layer seems to be learning something very interpretable and understandable. Specifically, Figure 3.5 shows the D matrix of our solution before and after learning (we permute the rows to make them ordered by the magnitude of where the large-absolute-value entries occurs). What is interesting in this picture is that the learned D matrix typically captures exactly the same intuition as the D matrix used by total variation denoising: a mainly sparse matrix with a few entries of alternating sign next to each other. This implies that for the data set we have, total variation denoising is indeed the "right" way to think about denoising the resulting signal, but if some other noise process were to generate the data, then we can learn that process instead. We can then attain lower actual error for the method (in this case similar though slightly higher than the TV solution), by fixing the learned sparsity of the D matrix and then fine tuning.

Fine-tuning and improving the total variation solution

To finally highlight the ability of the OptNet methods to *improve* upon the results of a convex program, specifically tailoring to the data. Here, we use the same OptNet architecture as in the previous subsection, but initialize D to be the differencing matrix as in

Method	Train MSE	Test MSE
FC Net	18.5	29.8
Pure OptNet	52.9	53.3
Total Variation	16.3	16.5
OptNet Tuned TV	13.8	**14.4**

Table 3.1: Denoising task error rates.

the total variation solution. As shown in Table 3.1, the procedure is able to improve both the training and testing MSE over the TV solution, specifically improving upon test MSE by 12%. Figure 3.6 shows the convergence of the OptNet fine-tuned TV solution.

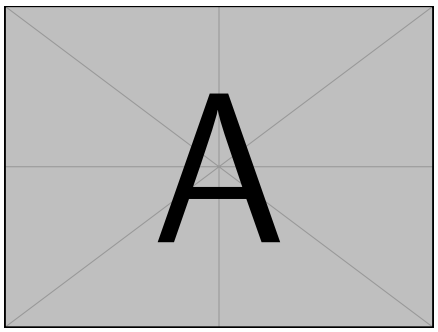


Figure 3.6: Error rate from fine-tuning the TV solution for denoising

3.4.3 MNIST

In this section we consider the integration of QP OptNet layers into a traditional fully connected network for the MNIST problem. The results here show only very marginal improvement if any over a fully connected layer (MNIST, after all, is very fairly well-solved by a fully connected network, let alone a convolution network). But our main point of this comparison is simply to illustrate that we can include these layers within existing network architectures and efficiently propagate the gradients through the layer.

Specifically we use a FC600-FC10-FC10-SoftMax fully connected network and compare it to a FC600-FC10-Optnet10-SoftMax network, where the numbers after each layer indicate the layer size. The OptNet layer in this case includes only inequality constraints and the previous layer is only used in the linear objective term $p(z_i) = z_i$. To keep $Q \succ 0$, we use a Cholesky factorization $Q = LL^T + \epsilon I$ and directly learn L (without any information from the previous layer). We also directly learn A and G, and to ensure a feasible solution always exists, we select some learnable z_0 and s_0 and set $b = Az_0$ and $h = Gz_0 + s_0$.

Figure 3.7 shows that the results are similar for both networks with slightly lower error and less variance in the OptNet network.

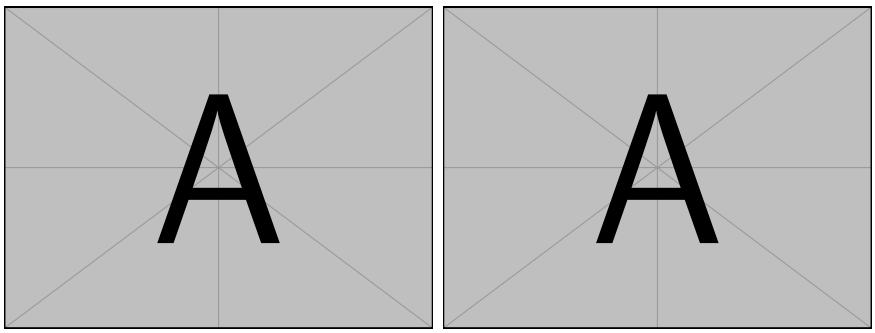


Figure 3.7: Training performance on MNIST; top: fully connected network; bottom: OptNet as final layer.)

			3
1			
		4	
4			1

2	4	1	3
1	3	2	4
3	1	4	2
4	2	3	1

Figure 3.8: Example mini-Sudoku initial problem and solution.

3.4.4 Sudoku

Finally, we present the main illustrative example of the representational power of our approach, the task of learning the game of Sudoku. Sudoku is a popular logical puzzle, where a (typically 9x9) grid of points must be arranged given some initial point, so that each row, each column, and each 3x3 grid of points must contain one of each number 1 through 9. We consider the simpler case of 4x4 Sudoku puzzles, with numbers 1 through 4, as shown in Section 3.4.3.

Sudoku is fundamentally a constraint satisfaction problem, and is trivial for computers to solve when told the rules of the game. However, if we do not know the rules of the game, but are only presented with examples of unsolved and the corresponding solved puzzle, this is a challenging task. We consider this to be an interesting benchmark task for algorithms that seek to capture complex strict relationships between all input and output variables. The input to the algorithm consists of a 4x4 grid (really a 4x4x4 tensor with a one-hot encoding for known entries an all zeros for unknown entries), and the desired output is a 4x4x4 tensor of the one-hot encoding of the solution.

This is a problem where traditional neural networks have difficulties learning the necessary hard constraints. As a baseline inspired by the models at `https://github.com/Kyubyong/sudoku`, we implemented a multilayer feedforward network to attempt to solve Sudoku problems. Specifically, we report results for a network that has 10 convolutional layers with 512 3x3 filters each, and tried other architectures as well. The OptNet layer we use on this task is a completely generic QP in "standard form" with only positivity inequality constraints but an arbitrary constraint matrix $Ax = b$, a small $Q = 0.1I$ to make sure the problem is strictly feasible, and with the linear term q simply being the input one-hot encoding of the Sudoku problem. We know that Sudoku *can* be approximated well

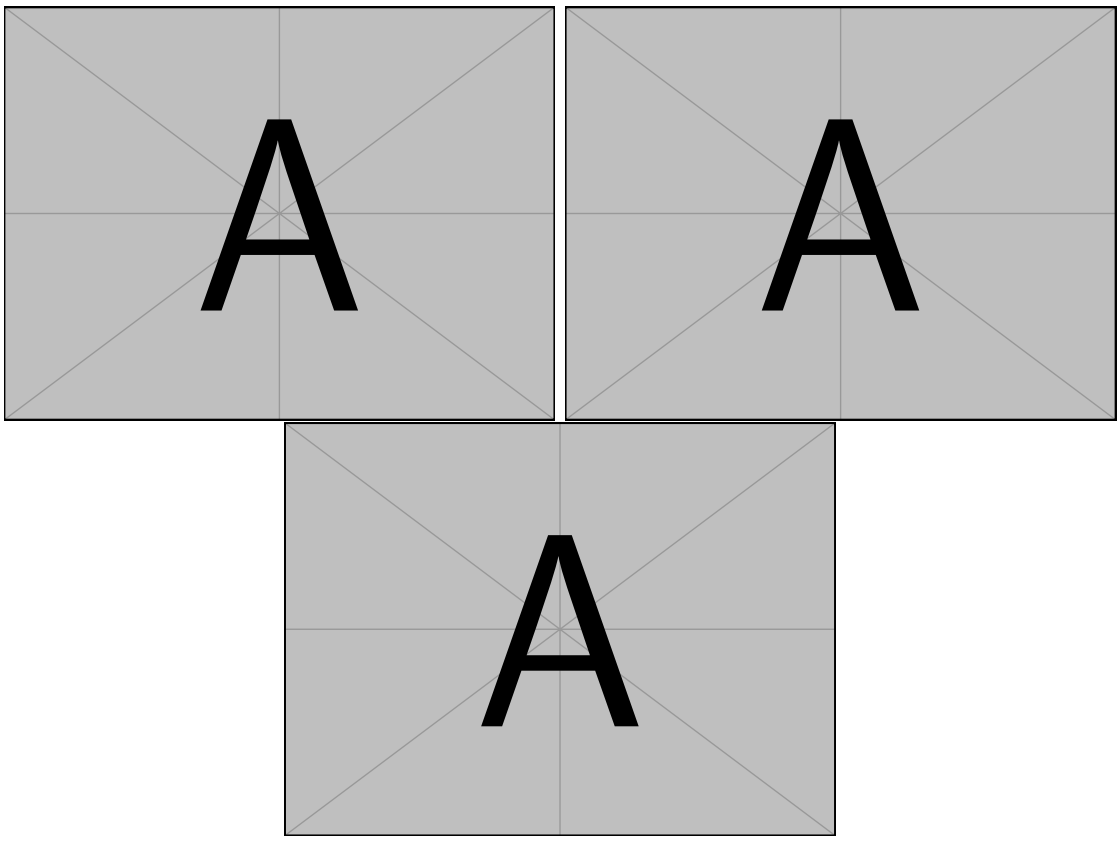


Figure 3.9: Sudoku training plots.

with a linear program (indeed, integer programming is a typical solution method for such problems), but the model here is told nothing about the rules of Sudoku.

We trained these models using ADAM [KB14] to minimize the MSE (which we refer to as "loss") on a dataset we created consisting of 9000 training puzzles, and we then tested the models on 1000 different held-out puzzles. The error rate is the percentage of puzzles solved correctly if the cells are assigned to whichever index is largest in the prediction. Figure 3.9 shows that the convolutional is able to learn all of the necessary logic for the task and ends up over-fitting to the training data. We contrast this with the performance of the OptNet network, which learns most of the correct hard constraints within the first three epochs and is able to generalize much better to unseen examples.

3.5 Conclusion

We have presented OptNet, a neural network architecture where we use optimization problems as a single layer in the network. We have derived the algorithmic formulation for differentiating through these layers, allowing for backpropagating in end-to-end architectures. We have also developed an efficient batch solver for these optimizations based upon a primal-dual interior point method, and developed a method for attaining the necessary

gradient information "for free" from this approach. Our experiments highlight the potential power of these networks, showing that they can solve problems where existing networks are very poorly suited, such as learning Sudoku problems purely from data. There are many future directions of research for these approaches, but we feel that they add another important primitive to the toolbox of neural network practitioners.

Chapter 4

Input-Convex Neural Networks

This chapter describes the input-convex neural network (ICNN) architecture that helps make inference and learning in deep energy-based models and structured prediction more tractable. These are scalar-valued (potentially deep) neural networks with constraints on the network parameters such that the output of the network is a convex function of (some of) the inputs. The networks allow for efficient inference via optimization over some inputs to the network given others, and can be applied to settings including structured prediction, data imputation, reinforcement learning, and others. In this chapter we lay the basic groundwork for these models, proposing methods for inference, optimization and learning, and analyze their representational power. We show that many existing neural network architectures can be made input-convex with a minor modification, and develop specialized optimization algorithms tailored to this setting. Finally, we highlight the performance of the methods on multi-label prediction, image completion, and reinforcement learning problems, where we show improvement over the existing state of the art in many cases.

The contents of this chapter have been previously published at ICML 2017 in Amos, Xu, and Kolter [AXK17].

4.1 Introduction

Input-convex neural networks (ICNNs) are scalar-valued (potentially deep) neural networks with constraints on the network parameters such that the output of the network is a convex function of (some of) the inputs. The networks allow for efficient inference via optimization over some inputs to the network given others, and can be applied to settings including structured prediction, data imputation, reinforcement learning, and others. In this chapter we lay the basic groundwork for these models, proposing methods for inference, optimization and learning, and analyze their representational power. We show that many existing neural network architectures can be made input-convex with a minor modification, and develop specialized optimization algorithms tailored to this setting. Finally, we highlight the performance of the methods on multi-label prediction, image completion, and reinforcement learning problems, where we show improvement over the existing state

of the art in many cases.

More specifically, input-convex neural networks are *scalar-valued* neural networks $f(x, y; \theta)$ where x and y denotes inputs to the function and θ denotes the parameters, built in such a way that the network is convex in (a subset of) *inputs* y.[1] The fundamental benefit to these ICNNs is that we can *optimize* over the convex inputs to the network given some fixed value for other inputs. That is, given some fixed x (and possibly some fixed elements of y) we can globally and efficiently (because the problem is convex) solve the optimization problem

$$\operatorname*{argmin}_y f(x, y; \theta). \tag{4.1}$$

Fundamentally, this formalism lets us perform inference in the network via *optimization*. That is, instead of making predictions in a neural network via a purely feedforward process, we can make predictions by optimizing a scalar function (which effectively plays the role of an energy function) over some inputs to the function given others. There are a number of potential use cases for these networks.

Structured prediction As is perhaps apparent from our notation above, a key application of this work is in structured prediction. Given (typically high-dimensional) structured input and output spaces $\mathcal{X} \times \mathcal{Y}$, we can build a network over (x, y) pairs that encodes the energy function for this pair, following typical energy-based learning formalisms [LeC+06]. Prediction involves finding the $y \in \mathcal{Y}$ that minimizes the energy for a given x, which is exactly the argmin problem in Equation (4.1). In our setting, assuming that $\mathcal{Y}$ is a convex space (a common assumption in structured prediction), this optimization problem is convex. This is similar in nature to the structured prediction energy networks (SPENs) [BM16], which also use deep networks over the input and output spaces, with the difference being that in our setting f is convex in y, so the optimization can be performed globally.

Data imputation Similar to structured prediction but slightly more generic, if we are given some space $\mathcal{Y}$ we can learn a network $f(y; \theta)$ (removing the additional x inputs, though these can be added as well) that, given an example with some subset $\mathcal{I}$ missing, imputes the likely values of these variables by solving the optimization problem as above $\hat{y}_\mathcal{I} = \operatorname{argmin}_{y_\mathcal{I}} f(y_\mathcal{I}, y_{\bar{\mathcal{I}}}; \theta)$ This could be used e.g., in image inpainting where the goal is to fill in some arbitrary set of missing pixels given observed ones.

Continuous action reinforcement learning Given a reinforcement learning problem with potentially continuous state and action spaces $\mathcal{S} \times \mathcal{A}$, we can model the (negative) Q function, $-Q(s, a; \theta)$ as an input convex neural network. In this case the action selection procedure can be formulated as a convex optimization problem $a^\star(s) = \operatorname{argmin}_a -Q(s, a; \theta)$.

[1] We emphasize the term "input convex" since convexity in machine learning typically refers to convexity (of the loss minimization learning problem) in the *parameters*, which is not the case here. Note that in our notation, f needs only be a convex function in y, and may still be non-convex in the remaining inputs x. Training these neural networks remains a nonconvex problem, and the convexity is only being exploited at inference time.

4.2 Connections to related work

Energy-based learning The interplay between inference, optimization, and structured prediction has a long history in neural networks. Several early incarnations of neural networks were explicitly trained to produce structured sequences (e.g. Simard and LeCun [SL91]), and there was an early appreciation that structured models like hidden Markov models could be combined with the outputs of neural networks [BLH94]. Much of this earlier work is surveyed and synthesized by LeCun, Chopra, Hadsell, Ranzato, and Huang [LeC+06], who give a tutorial on these energy based learning methods. In recent years, there has been a strong push to further incorporate structured prediction methods like conditional random fields as the "last layer" of a deep network architecture [PBX09; Zhe+15; Che+15a]. Several methods have proposed to build general neural networks over joint input and output spaces, and perform inference over outputs using generic optimization techniques such as Generative Adversarial Networks (GANs) [Goo+14] and Structured Prediction Energy Networks (SPENs) [BM16]. SPENs provide a deep structure over input and output spaces that performs the inference in Equation (4.1) as a non-convex optimization problem.

The current work is highly related to the past approaches but also differs in a particular way. Each of these structured prediction methods based upon energy-based models operates in one of two ways, either: 1) the architecture is built in a very particular way such that optimization over the output is guaranteed to be "easy" (e.g. convex, or the result of running some inference procedure), usually by introducing a structured linear objective at the last layer of the network; or 2) no attempt is made to make the architecture "easy" to run inference over, and instead a general model is built over the output space. In contrast, our approach lies somewhere in between: by ensuring convexity of the resulting decision space, we are constraining the inference problem to be easy in some respect, but we specify very little about the architecture other than the constraints required to make it convex. In particular, as we will show, the network architecture over the variables to be optimized over can be deep and involve multiple non-linearities. The goal of the proposed work is to allow for complex functions over the output without needing to specify them manually (exactly analogous to how current deep neural networks treat their input space).

Structured prediction and MAP inference Our work also draws some connection to MAP-inference-based learning and approximate inference. There are two broad classes of learning approaches in structured prediction: method that use probabilistic inference techniques (typically exploiting the fact that the gradient of log likelihood is given by the actual feature expectations minus their expectation under the learned model [KF09, Ch 20]), and methods that rely solely upon MAP inference (such as max-margin structured prediction [Tas+05; Tso+05]). MAP inference in particular also has close connections to optimization, as various convex relaxations of the general MAP inference problem often perform well in theory and practice.

The proposed methods can be viewed as an extreme case of these methods where inference is based *solely* upon a convex optimization problem that may not have any probabilistic semantics at all. Finally, although it is more abstract, we feel there is a philosophical similarity between our proposed approach and sum-product networks [PD11];

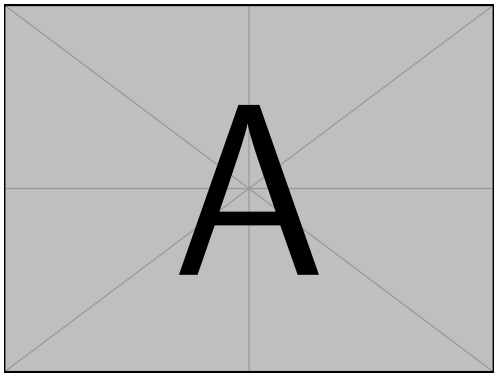


Figure 4.1: A fully input convex neural network (FICNN).

both settings define networks where inference is accomplished "easily" either by a sum-product message passing algorithm (by construction) or via convex optimization.

Fitting convex functions Finally, the proposed work relates to a topic less considered in the machine learning literature, that of fitting convex functions to data [BV04, pg. 338]. Indeed our learning problem can be viewed as parameter estimation under a model that is guaranteed to be convex by its construction. The most similar work of which we are aware specifically fits sums of rectified half-planes to data [MB09], which is similar to one layer of our rectified linear units. However, the actual training scheme is much different, and our deep network architecture allows for a much richer class of representations, while still maintaining convexity.

4.3 Convex neural network architectures

Here we more formally present different ICNN architectures and prove their convexity properties given certain constraints on the parameter space. Our chief claim is that the class of (full and partial) input convex models is rich and lets us capture complex joint models over the input to a network.

4.3.1 Fully input convex neural networks

To begin, we consider a fully convex, k-layer, fully connected ICNN that we call a FICNN and is shown in Figure 4.1. This model defines a neural network over the input y (i.e., omitting any x term in this function) using the architecture for $i = 0, \dots, k-1$

$$z_{i+1} = g_i \left(W_i^{(z)} z_i + W_i^{(y)} y + b_i \right), \quad f(y; \theta) = z_k \tag{4.2}$$

where z_i denotes the layer activations (with $z_0, W_0^{(z)} \equiv 0$), $\theta = \{W_{0:k-1}^{(y)}, W_{1:k-1}^{(z)}, b_{0:k-1}\}$ are the parameters, and g_i are non-linear activation functions. The central result on convexity of the network is the following:

Proposition 1. *The function f is convex in y provided that all $W_{1:k-1}^{(z)}$ are non-negative, and all functions g_i are convex and non-decreasing.*

The proof is simple and follows from the fact that non-negative sums of convex functions are also convex and that the composition of a convex and convex non-decreasing function is also convex (see e.g. Boyd and Vandenberghe [BV04, p. 3.2.4]). The constraint that the g_i be convex non-decreasing is not particularly restrictive, as current non-linear activation units like the rectified linear unit or max-pooling unit already satisfy this constraint. The constraint that the $W^{(z)}$ terms be non-negative is somewhat restrictive, but because the bias terms and $W^{(y)}$ terms can be negative, the network still has substantial representation power, as we will shortly demonstrate empirically.

One notable addition in the ICNN are the "passthrough" layers that directly connect the input y to hidden units in deeper layers. Such layers are unnecessary in traditional feedforward networks because previous hidden units can always be mapped to subsequent hidden units with the identity mapping; however, for ICNNs, the non-negativity constraint subsequent $W^{(z)}$ weights restricts the allowable use of hidden units that mirror the identity mapping, and so we explicitly include this additional passthrough. Some passthrough layers have been recently explored in the deep residual networks [He+15] and densely connected convolutional networks [HLW16], though these differ from those of an ICNN as they pass through hidden layers deeper in the network, whereas to maintain convexity our passthrough layers can only apply to the input directly.

Other linear operators like convolutions can be included in ICNNs without changing the convexity properties. Indeed, modern feedforward architectures such as AlexNet [KSH12], VGG [SZ14], and GoogLeNet [Sze+15] with ReLUs [NH10] can be made input convex with Proposition 1. In the experiments that follow, we will explore ICNNs with both fully connected and convolutional layers.

4.3.2 Convolutional input-convex architectures

Convolutional architectures are important for many vision tasks and can easily be made input-convex because the convolution is a linear operator. The construction of convolutional layers in ICNNs depends on the type of input and output space. If the input and output space are similarly structured (e.g. both spatial), the jth feature map of a convolutional PICNN layer i can be defined by

$$z_{i+1}^j = g_i\left(z_i * W_{i,j}^{(z)} + (Sx) * W_{i,j}^{(x)} + (Sy) * W_{i,j}^{(y)} + b_{i,j}\right) \tag{4.3}$$

where the convolution kernels W are the same size and S scales the input and output to be the same size as the previous feature map, and were we omit some of the Hadamard product terms that can appear above for simplicity of presentation.

If the input space is spatial, but the output space has another structure (e.g. the simplex), the convolution over the output space can be replaced by a matrix-vector operation, such as

$$z_{i+1}^j = g_i\left(z_i * W_{i,j}^{(z)} + (Sx) * W_{i,j}^{(x)} + B_{i,j}^{(y)} y + b_{i,j}\right) \tag{4.4}$$

where the product $B_{i,j}^{(y)} y$ is a scalar.

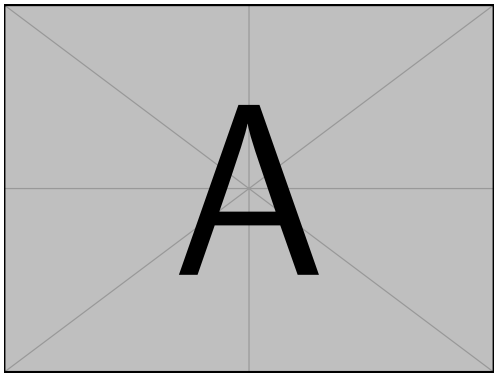


Figure 4.2: A partially input convex neural network (PICNN).

4.3.3 Partially input convex architectures

The FICNN provides joint convexity over the entire input to the function, which indeed may be a restriction on the allowable class of models. Furthermore, this full joint convexity is unnecessary in settings like structured prediction where the neural network is used to build a joint model over an input and output example space and only convexity over the outputs is necessary.

In this section we propose an extension to the pure FICNN, the partially input convex neural network (PICNN), that is convex over only some inputs to the network (in general ICNNs will refer to this new class). As we will show, these networks generalize both traditional feedforward networks and FICNNs, and thus provide substantial representational benefits. We define a PICNN to be a network over (x, y) pairs $f(x, y; \theta)$ where f is convex in y but not convex in x. Figure 4.2 illustrates one potential k-layer PICNN architecture defined by the recurrences

$$\begin{aligned} u_{i+1} &= \tilde{g}_i(\tilde{W}_i u_i + \tilde{b}_i) \\ z_{i+1} &= g_i\left(W_i^{(z)}\left(z_i \circ [W_i^{(zu)} u_i + b_i^{(z)}]_+\right) + \right. \\ &\qquad \left. W_i^{(y)}\left(y \circ (W_i^{(yu)} u_i + b_i^{(y)})\right) + W_i^{(u)} u_i + b_i\right) \\ f(x, y; \theta) &= z_k, \; u_0 = x \end{aligned} \tag{4.5}$$

where $u_i \in \mathbb{R}^{n_i}$ and $z_i \in \mathbb{R}^{m_i}$ denote the hidden units for the "x-path" and "y-path", where $y \in \mathbb{R}^p$, and where $\circ$ denotes the Hadamard product, the elementwise product between two vectors. The crucial element here is that unlike the FICNN, we only need the $W^{(z)}$ terms to be non-negative, and we can introduce arbitrary products *between* the u_i hidden units and the z_i hidden units. The following proposition highlights the representational power of the PICNN.

Proposition 2. *A PICNN network with k layers can represent any FICNN with k layers and any purely feedforward network with k layers.*

Proof. To recover a FICNN we simply set the weights over the entire x path to be zero and set $b^{(z)} = b^{(y)} = 1$. We can recover a feedforward network by noting that a traditional

feedforward network $\hat{f}(x;\theta)$ where $f : \mathcal{X} \to \mathcal{Y}$, can be viewed as a network with an inner product $f(x;\theta)^T y$ in its last layer (see e.g. LeCun, Chopra, Hadsell, Ranzato, and Huang [LeC+06] for more details). Thus, a feedforward network can be represented as a PICNN by setting the x path to be exactly the feedforward component, then having the y path be all zero except $W_{k-1}^{(yu)} = I$ and $W_{k-1}^{(y)} = 1^T$. □

4.4 Inference in ICNNs

Prediction in ICNNs (which we also refer to as inference), requires solving the convex optimization problem

$$\underset{y \in \mathcal{Y}}{\text{minimize}} \; f(x, y; \theta) \tag{4.6}$$

While the resulting tasks are convex optimization problems (and thus "easy" to solve in some sense), in practice this still involves the solution of a potentially very complex optimization problem. We discuss here several approaches for approximately solving these optimization problems. We can usually obtain reasonably accurate solutions in many settings using a procedure that only involves a small number of forward and backward passes through the network, and which thus has a complexity that is at most a constant factor worse than that for feedforward networks. The same consideration will apply to training such networks, which we will discuss in Section 4.5.

4.4.1 Exact inference in ICNNs

Although it is not a practical approach for solving the optimization tasks, we first highlight the fact that the inference problem for the networks presented above (where the non-linear are either ReLU or linear units) can be posed as as linear program. Specifically, considering the FICNN network in (4.2) can be written as the optimization problem

$$\begin{aligned} \underset{y, z_1, \ldots, z_k}{\text{minimize}} \;\; & z_k \\ \text{subject to} \;\; & z_{i+1} \geq W_i^{(z)} z_i + W_i^{(y)} y + b_i, \quad i = 0, \ldots, k-1 \\ & z_i \geq 0, \quad i = 1, \ldots, k-1. \end{aligned} \tag{4.7}$$

This problem exactly replicates the equations of the FICNN, with the exception that we have replaced ReLU and the equality constraint between layers with a positivity constraint on the z_i terms and an inequality. However, because we are minimizing the final z_k term, and because each inequality constraint is convex, at the solution one of these constraints must be tight, i.e., $(z_i)_j = (W_i^{(z)} z_i + W_i^{(y)} y + b_i)_j$ or $(z_i)_j = 0$, which recovers the ReLU non-linearity exactly. The exact same procedure can be used to write to create an exact inference procedure for the PICNN.

Although the LP formulation is appealing in its simplicity, in practice these optimization problems will have a number of variables equal to the *total* number of activations in the entire network. Furthermore, most LP solution methods to solve such problems require that we form *and invert* structured matrices with blocks such as $W_i^T W_i$ — the

case for most interior-point methods [Wri97] or even approximate algorithms such as the alternating direction method of multipliers [Boy+11] — which are large dense matrices or have structured forms such as non-cyclic convolutions that are expensive to invert. Even incremental approaches like the Simplex method require that we form inverses of subsets of columns of these matrices, which are additionally different for structured operations like convolutions, and which overall still involve substantially more computation than a single forward pass. Furthermore, such solvers typically do not exploit the substantial effort that has gone in to accelerating the forward and backward computation passes for neural networks using hardware such as GPUs. Thus, as a whole, these do not present a viable option for optimizing the networks.

4.4.2 Approximate inference in ICNNs

Because of the impracticality of exact inference, we focus on approximate approaches to optimizing over the inputs to these networks, but ideally ones that still exploit the convexity of the resulting problem. We specifically focus on gradient-based approaches, which use the fact that we can easily compute the gradient of an ICNN with respect to its inputs, $\nabla_y f(x, y; \theta)$, using backpropagation.

Gradient descent. The simplest gradient-based methods for solving Equation (4.6) is just (projected sub-) gradient descent, or modifications such as those that use a momentum term [Pol64; RHW88], or spectral step size modifications [BB88; BMR00]. That is, we start with some initial $\hat{y}$ and repeat the update

$$\hat{y} \leftarrow \mathcal{P}_{\mathcal{Y}} \left(\hat{y} - \alpha \nabla_y f(x, \hat{y}; \theta) \right) \tag{4.8}$$

This method is appealing in its simplicity, but suffers from the typical problems of gradient descent on non-smooth objectives: we need to pick a step size and possibly use a sequence of decreasing step sizes, and don't have an obvious method to assess how accurate of a current solution we have obtained (since an ICNN with ReLUs is piecewise linear, it will not have zero gradient at the solution). The method is also more challenging to integrate with some learning procedures, as we often need to differentiate through an entire chain of the gradient descent algorithm [Dom12]. Thus, while the method can sometimes work in practice, we have found that other approaches typically far outperform this method, and we will focus on alternative approximate approaches for the remainder of this section.

4.4.3 Approximate inference via the bundle method

We here review the basic bundle method [SVL08] that we build upon in our bundle entropy method. The bundle method takes advantage of the fact that for a convex objective, the first-order approximation at any point is a global *under-estimator* of the function; this lets us maintain a piecewise linear lower bound on the function by adding cutting planes formed by this first order approximation, and then repeatedly optimizing this lower bound. Specifically, the process follows the procedure shown in Algorithm 1. Denoting the iterates of the algorithm as y^k, at each iteration of the algorithm, we compute the first order

approximation to the function

$$f(x, y^k; \theta) + \nabla_y f(x, y^k; \theta)^T (y - y^k) \tag{4.9}$$

and update the next iteration by solving the optimization problem

$$y^{k+1} := \operatorname*{argmin}_{y \in \mathcal{Y}} \max_{1 \le i \le k} \{f(x, y^i; \theta) + \nabla_y f(x, y^i; \theta)^T (y - y^i)\}. \tag{4.10}$$

A bit more concretely, the optimization problem can be written via a set of linear inequality constraints

$$y^{k+1}, t^{k+1} := \operatorname*{argmin}_{y \in \mathcal{Y}, t} \{t \mid Gy + h \le t1\} \tag{4.11}$$

where $G \in \mathbb{R}^{k \times n}$ has rows equal to

$$g_i^T = \nabla_y f(x, y^i; \theta)^T \tag{4.12}$$

and $h \in \mathbb{R}^k$ has entries equal to

$$h_i = f(x, y^i; \theta) - \nabla_y f(x, y^i; \theta)^T y^i. \tag{4.13}$$

Algorithm 1 A typical bundle method to optimize $f : \mathbb{R}^{m \times n} \to \mathbb{R}$ over $\mathbb{R}^n$ for K iterations with a fixed x and initial starting point y^1.

function BUNDLEMETHOD(f, x, y^1, K)
  $G \leftarrow 0 \in \mathbb{R}^{K \times n}$
  $h \leftarrow 0 \in \mathbb{R}^K$
  for $k = 1, K$ **do**
    $G_k^T \leftarrow \nabla_y f(x, y^k; \theta)^T$ ▷ kth row of G
    $h_k \leftarrow f(x, y^k; \theta) - \nabla_y f(x, y^k; \theta)^T y^k$
    $y^{k+1}, t^{k+1} \leftarrow \operatorname{argmin}_{y \in \mathcal{Y}, t} \{t \mid G_{1:k} y + h_{1:k} \le t1\}$
  end for
  return y^{K+1}
end function

4.4.4 Approximate inference via the bundle entropy method

An alternative approach to gradient descent is the bundle method [SVL08], also known as the epigraph cutting plane approach, which iteratively optimizes a piecewise lower bound on the function given by the maximum over a set of first-order approximations. However, as, the traditional bundle method is not well suited to our setting (we need to evaluate a number of gradients equal to the dimension of x, and solve a complex optimization problem at each step) we have developed a new optimization algorithm for this domain that we term the *bundle entropy method*. This algorithm specifically applies to the (common) case where $\mathcal{Y}$ is bounded, which we assume to be $\mathcal{Y} = [0, 1]^n$ (other upper or lower bounds can

be attained through scaling). The method is also easily extensible to the setting where elements of $\mathcal{Y}$ belong to a higher-dimensional probability simplex as well.

For this approach, we consider adding an additional "barrier" function to the optimization in the form of the negative entropy $-H(y)$, where

$$H(y) = -\sum_{i=1}^{n}(y_i \log y_i + (1-y_i)\log(1-y_i)). \tag{4.14}$$

In other words, we instead want to solve the optimization problem $\operatorname{argmin}_y \; f(x,y;\theta) - H(y)$ (with a possible additional scaling term). The negative entropy is a convex function, with the limits of $\lim_{y\to 0} H(y) = \lim_{y\to 1} H(y) = 0$, and negative values in the interior of this range. The function acts as a barrier because, although it does not approach infinity as it reaches the barrier of the feasible set, its gradient *does* approach infinity as it reaches the barrier, and thus the optimal solution will always lie in the interior of the unit hypercube $\mathcal{Y}$.

An appealing feature of the entropy regularization comes from its close connection with sigmoid units in typical neural networks. It follows easily from first-order optimality conditions that the optimization problem

$$\underset{y}{\text{minimize}} \; c^T y - H(y) \tag{4.15}$$

is given by $y^\star = 1/(1+\exp(c))$. Thus if we consider the "trivial" PICNN mentioned in Section 4.3.3, which simply consists of the function $f(x,y;\theta) = y^T \tilde{f}(x;\theta)$ for some purely feedforward network $\tilde{f}(x;\theta)$, then the entropy-regularized minimization problem gives a solution that is equivalent to simply taking the sigmoid of the neural network outputs. Thus, the move to ICNNs can be interpreted as providing a more structured joint energy functional over the linear function implicitly used by sigmoid layers.

At each iteration of the bundle entropy method, we solve the optimization problem

$$y^{k+1}, t^{k+1} := \underset{y,t}{\operatorname{argmin}} \; \{t - H(y) \mid Gy + h \le t1\} \tag{4.16}$$

where $G \in \mathbb{R}^{k\times n}$ has rows equal to

$$g_i^T = \nabla_y f(x, y^i; \theta)^T \tag{4.17}$$

and $h \in \mathbb{R}^k$ has entries equal to

$$h_i = f(x, y^i; \theta) - \nabla_y f(x, y^i; \theta)^T y^i. \tag{4.18}$$

The Lagrangian of the optimization problem is

$$\mathcal{L}(y,t,\lambda) = t - H(y) + \lambda^T(Gy + h - t1) \tag{4.19}$$

and differentiating with respect to y and t gives the optimality conditions

$$\begin{aligned} \nabla_y \mathcal{L}(y,t,\lambda) = 0 &\implies y = \frac{1}{1+\exp(G^T\lambda)} \\ \nabla_t \mathcal{L}(y,t,\lambda) = 0 &\implies 1^T\lambda = 1 \end{aligned} \tag{4.20}$$

which in turn leads to the dual problem

$$\begin{aligned} \underset{\lambda}{\text{maximize}} \;\; & (G1 + h)^T \lambda - 1^T \log(1 + \exp(G^T \lambda)) \\ \text{subject to} \;\; & \lambda \geq 0, 1^T \lambda = 1. \end{aligned} \tag{4.21}$$

This is a smooth optimization problem over the unit simplex, and can be solved using a method like the Projected Newton method of [Ber82, pg. 241, eq. 97]. A complete description of the bundle entropy method is given in Algorithm 2. For lower dimensional problems, the bundle entropy method often attains an exact solution after a relatively small number of iterations. And even for larger problems, we find that the approximate solutions generated by a very small number of iterations (we typically use 5 iterations), still substantially outperform gradient descent approaches. Further, because we maintain an explicit lower bound on the function, we can compute an optimality gap of our solution, though in practice just using a fixed number of iterations performs well.

Algorithm 2 Our bundle entropy method to optimize $f : \mathbb{R}^m \times [0,1]^n \to \mathbb{R}$ over $[0,1]^n$ for K iterations with a fixed x and initial starting point y^1.

function BundleEntropyMethod(f, x, y^1, K)
  $G_\ell \leftarrow [\,]$
  $h_\ell \leftarrow [\,]$
  for $k = 1, K$ **do**
    Append(G_ℓ, $\nabla_y f(x, y^k; \theta)^T$)
    Append(h_ℓ, $f(x, y^k; \theta) - \nabla_y f(x, y^k; \theta)^T y^k$)
    $a_k \leftarrow$ Length(G_ℓ) ▷ The number of active constraints.
    $G_k \leftarrow$ Concat(G_ℓ)$\in \mathbb{R}^{a_k \times n}$
    $h_k \leftarrow$ Concat(h_ℓ)$\in \mathbb{R}^{a_k}$
    if $a_k = 1$ **then**
      $\lambda_k \leftarrow 1$
    else
      $\lambda_k \leftarrow$ ProjNewtonLogistic(G_k, h_k)
    end if
    $y^{k+1} \leftarrow (1 + \exp(G_k^T \lambda_k))^{-1}$
    Delete($G_\ell[i]$ and $h_\ell[i]$ where $\lambda_i \leq 0$) ▷ Prune inactive constraints.
  end for
  return y^{K+1}
end function

4.5 Learning in ICNNs

Generally speaking, ICNN learning shapes the objective's energy function to produce the desired values when optimizing over the relevant inputs. That is, for a given input output pair $(x, y^\star)$, our goal is to find ICNN parameters θ such that

$$y^\star \approx \underset{y}{\text{argmin}}\, \tilde{f}(x, y; \theta) \tag{4.22}$$

where for the entirely of this section, we use the notation $\tilde{f}$ to denote the combination of the neural network function *plus* the regularization term such as $-H(y)$, if it is included, i.e.

$$\tilde{f}(x, y; \theta) = f(x, y; \theta) - H(y). \tag{4.23}$$

Although we only discuss the entropy regularization in this work, we emphasize that other regularizers are also possible. Depending on the setting, there are several different approaches we can use to ensure that the ICNN achieves the desired targets, and we consider three approaches below: direct functional fitting, max-margin structured prediction, and argmin differentiation.

Direct functional fitting. We first note that in some domains, we do not need a specialized procedure for fitting ICNNs, but can use existing approaches that directly fit the ICNN. An example of this is the Q-learning setting. Given some observed tuple (s, a, r, s'), Q learning updates the parameters θ with the gradient

$$\left(Q(s, a) - r - \gamma \max_{a'} Q(s', a')\right) \nabla_\theta Q(s, a), \tag{4.24}$$

where the maximization step is carried out with gradient descent or the bundle entropy method. These updates can be applied to ICNNs with the only additional requirement that we project the weights onto their feasible sets after this update (i.e., clip or project any W terms that are required to be positive). Algorithm 3 gives a complete description of deep Q-learning with ICNNs.

4.5.1 Max-margin structured prediction

In the more traditional structured prediction setting, where we do not aim to fit the energy function directly but fit the predictions made by the system to some target outputs, there are different possibilities for learning the ICNN parameters. One such method is based upon the max-margin structured prediction framework [Tso+05; Tas+05]. Given some training example $(x, y^\star)$, we would like to require that this example has a joint energy that is lower than all other possible values for y. That is, we want the function $\tilde{f}$ to satisfy the constraint

$$\tilde{f}(x, y^\star; \theta) \leq \min_y \tilde{f}(x, y; \theta) \tag{4.25}$$

Unfortunately, these conditions can be trivially fit by choosing a constant $\tilde{f}$ (although the entropy term alleviates this problem slightly, we can still choose an approximately constant function), so instead the max-margin approach adds a margin-scaling term that requires this gap to be larger for y further from $y^\star$, as measured by some loss function $\Delta(y, y^\star)$. Additionally adding slack variables to allow for potential violation of these constraints, we arrive at the typical max-margin structured prediction optimization problem

$$\begin{aligned} \underset{\theta, \xi \geq 0}{\text{minimize}} \quad & \frac{\lambda}{2}\|\theta\|_2^2 + \sum_{i=1}^m \xi_i \\ \text{subject to} \quad & \tilde{f}(x_i, y_i; \theta) \leq \min_{y \in \mathcal{Y}} \left(\tilde{f}(x_i, y; \theta) - \Delta(y_i, y)\right) - \xi_i \end{aligned} \tag{4.26}$$

As a simple example, for multiclass classification tasks where $y^\star$ denotes a "one-hot" encoding of examples, we can use a multi-variate entropy term and let $\Delta(y, y^\star) = y^{\star T}(1-y)$. Training requires solving this "loss-augmented" inference problem, which is convex for suitable choices of the margin scaling term.

The optimization problem (4.26) is naturally still *not convex* in θ, but can be solved via the subgradient method for structured prediction [RBZ07]. This algorithm iteratively selects a training example x_i, y_i, then 1) solves the optimization problem

$$y^\star = \operatorname*{argmin}_{y \in \mathcal{Y}} f(x_i, y; \theta) - \Delta(y_i, y) \tag{4.27}$$

and 2) if the margin is violated, updates the network's parameters according to the subgradient

$$\theta := \mathcal{P}_+ \left[\theta - \alpha \left(\lambda\theta + \nabla_\theta f(x_i, y_i, \theta) - \nabla_\theta f(x_i, y^\star; \theta)\right)\right] \tag{4.28}$$

where $\mathcal{P}_+$ denotes the projection of $W^{(z)}_{1:k-1}$ onto the non-negative orthant. This method can be easily adapted to use mini-batches instead of a single example per subgradient step, and also adapted to alternative optimization methods like AdaGrad [DHS11] or ADAM [KB14]. Further, a fast approximate solution to $y^\star$ can be used instead of the exact solution.

4.5.2 Argmin differentiation

In our final proposed approach, that of argmin differentiation, we propose to directly minimize a loss function between true outputs and the outputs predicted by our model, where these predictions themselves are the result of an optimization problem. We explicitly consider the case where the approximate solution to the inference problem is attained via the previously-described bundle entropy method, typically run for some fixed (usually small) number of iterations. To simplify notation, in the following we will let

$$\begin{aligned} \hat{y}(x; \theta) &= \operatorname*{argmin}_y \min_t \ \{t - H(y) \mid Gy + h \leq t1\} \\ &\approx \operatorname*{argmin}_y \tilde{f}(x, y; \theta) \end{aligned} \tag{4.29}$$

refer to the *approximate* minimization over y that results from running the bundle entropy method, specifically at the last iteration of the method.

Given some example $(x, y^\star)$, our goal is to compute the gradient, with respect to the ICNN parameters, of the loss between $y^\star$ and $\hat{y}(x; \theta)$: $\ell(\hat{y}(x; \theta), y^\star)$. This is in some sense the most direct analogue to traditional neural network learning, since we typically optimize networks by minimizing some loss between the network's (feedforward) predictions and the true desired labels. Doing this in the predictions-via-optimization setting requires that we differentiate "through" the argmin operator, which can be accomplished via implicit differentiation of the KKT optimality conditions. Although the derivation is somewhat involved, the final result is fairly compact, and is given by the following proposition (for simplicity, we will write $\hat{y}$ below instead of $\hat{y}(x; \theta)$ when the notation should be clear):

Proposition 3. *The gradient of the neural network loss for predictions generated through the minimization process is*

$$\nabla_\theta \ell(\hat{y}(x;\theta), y^\star) = \sum_{i=1}^{k} (c_i^\lambda \nabla_\theta f(x, y^i; \theta) + \nabla_\theta \left(\nabla_y f(x, y^i; \theta)^T \left(\lambda_i c^y + c_i^\lambda \left(\hat{y}(x;\theta) - y^i \right) \right) \right)) \tag{4.30}$$

where y^i denotes the solution returned by the ith iteration of the entropy bundle method, λ denotes the dual variable solution of the entropy bundle method, and where the c variables are determined by the solution to the linear system

$$\begin{bmatrix} H & G^T & 0 \\ G & 0 & -1 \\ 0 & -1^T & 0 \end{bmatrix} \begin{bmatrix} c^y \\ c^\lambda \\ c^t \end{bmatrix} = \begin{bmatrix} -\nabla_{\hat{y}} \ell(\hat{y}, y^\star) \\ 0 \\ 0 \end{bmatrix}. \tag{4.31}$$

where $H = \operatorname{diag}\left(\frac{1}{\hat{y}} + \frac{1}{1-\hat{y}} \right)$.

Proof (of Proposition 3). We have by the chain rule that

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial \hat{y}} \left(\frac{\partial \hat{y}}{\partial G} \frac{\partial G}{\partial \theta} + \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial \theta} \right). \tag{4.32}$$

The challenging terms to compute in this equation are the $\frac{\partial \hat{y}}{\partial G}$ and $\frac{\partial \hat{y}}{\partial h}$ terms. These can be computed (although we will ultimately not compute them explicitly, but just compute the product of these matrices and other terms in the Jacobian), by implicit differentiation of the KKT conditions. Specifically, the KKT conditions of the bundle entropy method (considering only the active constraints at the solution) are given by

$$\begin{aligned} 1 + \log \hat{y} - \log(1 - \hat{y}) + G^T \lambda &= 0 \\ G\hat{y} + h - t1 &= 0 \\ 1^T \lambda &= 1. \end{aligned} \tag{4.33}$$

For simplicity of presentation, we consider first the Jacobian with respect to h. Taking differentials of these equations with respect to h gives

$$\begin{aligned} \operatorname{diag}\left(\frac{1}{\hat{y}} + \frac{1}{1-\hat{y}} \right) \mathsf{d}y + G^T \mathsf{d}\lambda &= 0 \\ G\mathsf{d}y + \mathsf{d}h - \mathsf{d}t 1 &= 0 \\ 1^T \mathsf{d}\lambda &= 0 \end{aligned} \tag{4.34}$$

or in matrix form

$$\begin{bmatrix} \operatorname{diag}\left(\frac{1}{\hat{y}} + \frac{1}{1-\hat{y}} \right) & G^T & 0 \\ G & 0 & -1 \\ 0 & -1^T & 0 \end{bmatrix} \begin{bmatrix} \mathsf{d}y \\ \mathsf{d}\lambda \\ \mathsf{d}t \end{bmatrix} = \begin{bmatrix} 0 \\ -\mathsf{d}h \\ 0 \end{bmatrix}. \tag{4.35}$$

To compute the Jacobian $\frac{\partial \hat{y}}{\partial h}$ we can solve the system above with the right hand side given by $\mathsf{d}h = I$, and the resulting $\mathsf{d}y$ term will be the corresponding Jacobian. However, in our ultimate objective we always left-multiply the proper terms in the above equation by $\frac{\partial \ell}{\partial \hat{y}}$. Thus, we instead define

$$\begin{bmatrix} c^y \\ c^\lambda \\ c^t \end{bmatrix} = \begin{bmatrix} \operatorname{diag}\left(\frac{1}{\hat{y}} + \frac{1}{1-\hat{y}}\right) & G^T & 0 \\ G & 0 & -1 \\ 0 & -1^T & 0 \end{bmatrix}^{-1} \begin{bmatrix} -(\frac{\partial \ell}{\partial \hat{y}})^T \\ 0 \\ 0 \end{bmatrix} \tag{4.36}$$

and we have the the simple formula for the Jacobian product

$$\frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} = (c^\lambda)^T. \tag{4.37}$$

A similar set of operations taking differentials with respect to G leads to the matrix equations

$$\begin{bmatrix} \operatorname{diag}\left(\frac{1}{\hat{y}} + \frac{1}{1-\hat{y}}\right) & G^T & 0 \\ G & 0 & -1 \\ 0 & -1^T & 0 \end{bmatrix} \begin{bmatrix} \mathsf{d}y \\ \mathsf{d}\lambda \\ \mathsf{d}t \end{bmatrix} = \begin{bmatrix} -\mathsf{d}G^T\lambda \\ -\mathsf{d}Gy \\ 0 \end{bmatrix} \tag{4.38}$$

and the corresponding Jacobian products / gradients are given by

$$\frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial G} = c^y \lambda^T + \hat{y}(c^\lambda)^T. \tag{4.39}$$

Finally, using the definitions that

$$g_i^T = \nabla_y f(x, y^i; \theta)^T, \quad h_i = f(x, y^k; \theta) - \nabla_y f(x, y^i; \theta)^T y^i \tag{4.40}$$

we recover the formula presented in the proposition. □

The complexity of computing this gradient will be linear in k, which is the number of *active* constraints at the solution of the bundle entropy method. The inverse of this matrix can also be computed efficiently by just inverting the $k \times k$ matrix $GH^{-1}G^T$ via a variable elimination procedure, instead of by inverting the full matrix. The gradients $\nabla_\theta f(x, y_i; \theta)$ are standard neural network gradients, and further, can be computed in the same forward/backward pass as we use to compute the gradients for the bundle entropy method. The main challenge of the method is to compute the terms of the form $\nabla_\theta(\nabla_y f(x, y_i; \theta)^T v)$ for some vector v. This quantity can be computed by most autodifferentiation tools (the gradient inner product $\nabla_y f(x, y_i; \theta)^T v$ itself just becomes a graph computation than can be differentiated itself), or it can be computed by a finite difference approximation. The complexity of computing this entire gradient is a small constant multiple of computing k gradients with respect to θ.

Given this ability to compute gradients with respect to an arbitrary loss function, we can fit the parameter using traditional stochastic gradient methods examples. Specifically, given an example (or a minibatch of examples) x_i, y_i, we compute gradients $\nabla_\theta \ell(\hat{y}(x_i; \theta), y_i)$ and update the parameters using e.g. the ADAM optimizer [KB14].

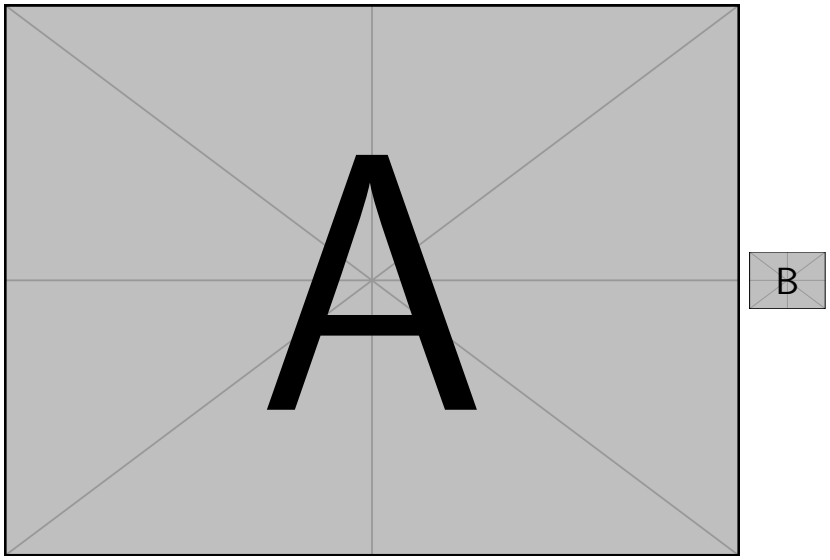


Figure 4.3: FICNN (top) and PICNN (bottom) classification of synthetic non-convex decision boundaries. Best viewed in color.

4.6 Experiments

Our experiments study the representational power of ICNNs to better understand the interplay between the model's restrictiveness and accuracy. Specifically, we evaluate the method on multi-label classification on the BibTeX dataset [KTV08], image completion using the Olivetti face dataset [SH94], and continuous action reinforcement learning in the OpenAI Gym [Bro+16]. We show that the methods compare favorably to the state of the art in many situations. The full source code for all experiments is available in the `icml2017` branch at `https://github.com/locuslab/icnn` and our implementation is built using Python [VD95] with the numpy [Oli06] and TensorFlow [Aba+16] packages.

4.6.1 Synthetic 2D example

We begin with a simple example to illustrate the classification performance of a two-hidden-layer FICNN and PICNN on two-dimensional binary classification tasks from the scikit-learn toolkit [Ped+11]. Figure 4.3 shows the classification performance on the dataset. The FICNN's energy function which is fully convex in $\mathcal{X} \times \mathcal{Y}$ jointly is able to capture complex, but sometimes restrictive decision boundaries. The PICNN, which is nonconvex over $\mathcal{X}$ but convex over $\mathcal{Y}$ overcomes these restrictions and can capture more complex decision boundaries.

4.6.2 Multi-Label Classification

We first study how ICNNs perform on multi-label classification with the BibTeX dataset and benchmark presented in Katakis, Tsoumakas, and Vlahavas [KTV08]. This benchmark

Method	Test Macro-F1
Feedforward net	0.396
ICNN	0.415
SPEN [BM16]	**0.422**

Table 4.1: Comparison of approaches on BibTeX multi-label classification task. (Higher is better.)

maps text classification from an input space $\mathcal{X}$ of 1836 bag-of-works indicator (binary) features to an output space $\mathcal{Y}$ of 159 binary labels. We use the train/test split of 4880/2515 from [KTV08] and evaluate with the macro-F1 score (higher is better). We use the ARFF version of this dataset from Mulan [Tso+11]. Our PICNN architecture for multi-label classification uses fully-connected layers with ReLU activation functions and batch normalization [IS15] along the input path. As a baseline, we use a fully-connected neural network with batch normalization and ReLU activation functions. Both architectures have the same structure (600 fully connected, 159 (#labels) fully connected). We optimize our PICNN with 30 iterations of gradient descent with a learning rate of 0.1 and a momentum of 0.3.

Table 4.1 compares several different methods for this problem. Our PICNN's final macro-F1 score of 0.415 outperforms our baseline feedforward network's score of 0.396, which indicates PICNNs have the power to learn a robust structure over the output space. SPENs obtain a macro-F1 score of 0.422 on this task [BM16] and pose an interesting comparison point to ICNNs as they have a similar (but not identical) deep structure that is non-convex over the input space. The difference of 0.007 between ICNNs and SPENs could be due to differences in our experimental setups, architectures, and random experimental noise. Figure 4.4 shows the training progress of the feed-forward and PICNN models.

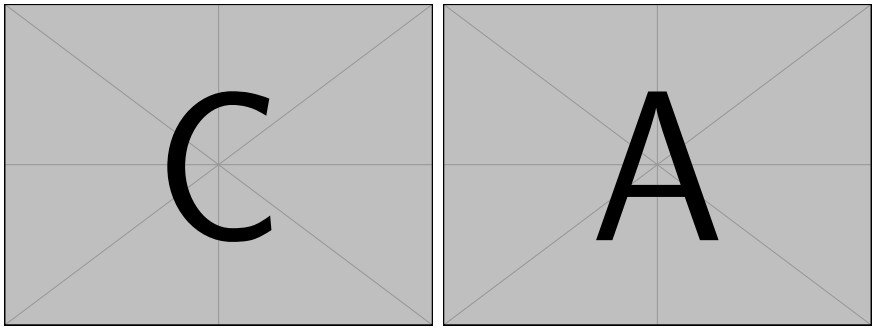


Figure 4.4: Training (blue) and test (red) macro-F1 score of a feedforward network (left) and PICNN (right) on the BibTeX multi-label classification dataset. The final test F1 scores are 0.396 and 0.415, respectively. (Higher is better.)

Figure 4.5: Example Olivetti test set image completions of the bundle entropy ICNN.

4.6.3 Image completion on the Olivetti faces

As a test of the system on a structured prediction task over a much more complex output space $\mathcal{Y}$, we apply a convolutional PICNN to face completion on the sklearn version [Ped+11] of the Olivetti data set [SH94], which contains 400 64x64 grayscale images. ICNNs for face completion should be invariant to translations and other transformations in the input space. To achieve this invariance, our PICNN is inspired by the DQN architecture in Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, et al. [Mni+15], which preserves this invariance in the different context of reinforcement learning. Specifically, our network is over (x, y) pairs where x (32x64) is the left half and y (32x64) is the right half of the image. The input and output paths are: 32x8x8 conv (stride 4x2), 64x4x4 conv (stride 2x2), 64x3x3 conv, 512 fully connected.

This experiment uses the same training/test splits and minimizes the mean squared error (MSE) as in Poon and Domingos [PD11]. We report the sum-product network results from Poon and Domingos [PD11] and have also implemented dilated CNN [YK15] and fully convolutional network (FCN) [LSD15] baselines. We also explore the tradeoffs between the bundle entropy method and gradient descent and compare to the non-convex variant to better understand the impacts of convexity. We use a learning rate of 0.01 and momentum of 0.9 with gradient descent for the inner optimization in the ICNN.

Table 4.2 shows the test MSEs for the different approaches and example image completions are shown in Figure 4.5. We note that as future work, an ICNN variant of the baseline dilated CNN and FCN architectures could be made in addition to the DQN architecture the ICNN in this experiment uses. For ICNNs, these results show that the bundle entropy method can leverage more information from these five iterations than gradient descent, even when the convexity constraint is relaxed. The PICNN trained with back-optimization with the relaxed convexity constraint slightly outperforms the network with the convexity constraint, but not the network trained with the bundle-entropy method. This shows that for image completion with PICNNs, convexity does not seem to inhibit the representational power. Furthermore, this experiment suggests that a small number of inner optimization iterations (five in this case) is sufficient for good performance.

Method	MSE
Sum-Product Network Baseline [PD11]	942.0
Dilated CNN Baseline [YK15]	800.0
FCN Baseline [LSD15]	**795.4**
ICNN - Bundle Entropy	833.0
ICNN - Gradient Decent	872.0
ICNN - Nonconvex	850.9

Table 4.2: Olivetti image completion test reconstruction errors.

4.6.4 Continuous Action Reinforcement Learning

Finally, we present standard benchmarks in continuous action reinforcement learning from the OpenAI Gym [Bro+16] that use the MuJoCo physics simulator [TET12]. We consider the environments shown in Table 4.3. We model the (negative) Q function, $-Q(s, a; \theta)$ as an ICNN and select actions with the convex optimization problem $a^\star(s) = \text{argmin}_a -Q(s, a; \theta)$. We use Q-learning to optimize the ICNN as described in Section 4.5 and Algorithm 3. At test time, the policy is selected by optimizing $Q(s, a; \theta)$. All of our experiments use a PICNN with two fully-connected layers that each have 200 hidden units. We compare to Deep Deterministic Policy Gradient (DDPG) [Lil+15] and Normalized Advantage Functions (NAF) [Gu+16b] as state-of-the-art off-policy learning baselines.[2]

Environment	# State	# Action
InvertedPendulum-v1	4	1
InvertedDoublePendulum-v1	11	1
Reacher-v1	11	2
HalfCheetah-v1	17	6
Swimmer-v1	8	2
Hopper-v1	11	3
Walker2d-v1	17	6
Ant-v1	111	8
Humanoid-v1	376	17
HumanoidStandup-v1	376	17

Table 4.3: State and action space sizes in the OpenAI gym MuJoCo benchmarks.

[2]Because there are not official DDPG or NAF implementations or results on the OpenAI gym tasks, we use the Simon Ramstedt's DDPG implementation from https://github.com/SimonRamstedt/ddpg and have re-implemented NAF.

Task	DDPG	NAF	ICNN
Ant	1000.00	999.03	**1056.29**
HalfCheetah	2909.77	2575.16	**3822.99**
Hopper	**1501.33**	1100.43	831.00
Humanoid	524.09	**5000.68**	433.38
HumanoidStandup	134265.96	116399.05	**141217.38**
InvDoubPend	**9358.81**	**9359.59**	**9359.41**
InvPend	**1000.00**	**1000.00**	**1000.00**
Reacher	-6.10	-6.31	**-5.08**
Swimmer	49.79	**69.71**	64.89
Walker2d	**1604.18**	1007.25	298.21

Table 4.4: Maximum test reward for ICNN algorithm versus alternatives on several OpenAI Gym tasks. (All tasks are v1.)

Algorithm 3 Deep Q-learning with ICNNs. `Opt-Alg` is a convex minimization algorithm such as gradient descent or the bundle entropy method. $\tilde{Q}_\theta$ is the objective the optimization algorithm solves. In gradient descent, $\tilde{Q}_\theta(s,a) = Q(s,a|\theta)$ and with the bundle entropy method, $\tilde{Q}_\theta(s,a) = Q(s,a|\theta) + H(a)$.

Select a discount factor $\gamma \in (0,1)$ and moving average factor $\tau \in (0,1)$
Initialize the ICNN $-Q(s,a|\theta)$ with target network parameters $\theta' \leftarrow \theta$ and a replay buffer $R \leftarrow \emptyset$
for each episode $e = 1, E$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for $i = 1, I$ **do**
    $a_i \leftarrow$ Opt-Alg$(-Q_\theta, s_i, a_{i,0}) + \mathcal{N}_i$ ▷ For some initial action $a_{i,0}$
    Execute a_i and observe r_{i+1} and s_{i+1}
    Insert$(R, (s_i, a_i, s_{i+1}, r_{i+1}))$
    Sample a random minibatch from the replay buffer: $R_M \subseteq R$
    for $(s_m, a_m, s_m^+, r_m^+) \in R_M$ **do**
      $a_m^+ \leftarrow$ Opt-Alg$(-Q_{\theta'}, s_m^+, a_{m,0}^+)$ ▷ Uses the target parameters θ'
      $y_m \leftarrow r_m^+ + \gamma Q(s_m^+, a_m^+|\theta')$
    end for
    Update θ with a gradient step to minimize $\mathcal{L} = \frac{1}{|R_M|} \sum_m \left(\tilde{Q}(s_m, a_m|\theta) - y_m\right)^2$
    $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ ▷ Update the target network.
  end for
end for

Table 4.4 shows the maximum test reward achieved on these tasks and, shows the ICNNs *can* be used as a drop-in replacement for a function approximator in Q-learning. Comparing the performance of the algorithms does not give a clear winner, as no algorithm strictly outperforms the others and there are non-deterministic and high-variance issues in

evaluating deep RL agents [Hen+18].

NAF poses a particularly interesting comparison point to ICNNs. In particular, NAF decomposes the Q function in terms of the value function an an advantage function $Q(s, a) = V(s) + A(s, a)$ where the advantage function is restricted to be *concave quadratic* in the actions, and thus always has a closed-form solution. In a sense, this closely mirrors the setup of the PICNN architecture: like NAF, we have a separate non-convex path for the s variables, and an overall function that is convex in a; however, the distinction is that while NAF requires that the convex portion be quadratic, the ICNN architecture allows any convex functional form.

4.7 Conclusion and future work

This chapter laid the groundwork for the input convex neural network model. By incorporating relatively simple constraints into existing network architectures, we can fit very general convex functions and the apply optimization as an inference procedure. Since many existing models already fit into this overall framework (e.g., CRF models perform an optimization over an output space where parameters are given by the output of a neural network), the proposed method presents an extension where the entire inference procedure is "learned" along with the network itself, without the need for explicitly building typical structured prediction architectures. This work explored only a small subset of the possible applications of these network, and the networks offer promising directions for many additional domains.

Part II

Extensions and Applications

Chapter 5

Differentiable `cvxpy` Optimization Layers

In this chapter, we show how to turn the `cvxpy` modeling language [DB16] into a differentiable optimization layer and implement our method in PyTorch [Pas+17b]. This allows users to express convex optimization layers in the intuitive `cvxpy` modeling language without needing to manually implement the backward pass.

5.1 Introduction

This thesis has presented differentiable optimization layers as a powerful class of operations for end-to-end learning that allow more specialized domain knowledge to be integrated into the modeling pipeline in a differentiable way. Convex optimization layers can be represented as

$$z_{i+1} = \underset{z}{\text{argmin}}\, f_\theta(z, z_i) \text{ s.t. } z \in \mathcal{C}_\theta(z_i) \tag{5.1}$$

where z_i is the previous layer, f is a convex objective function parameterized by θ, and $\mathcal{C}$ is a convex constraint set. From the perspective of end-to-end learning, convex optimization layers can be seen as a module that outputs z_{i+1} and has parameters θ that can be learned with gradient descent. We note that the convex case captures many of the applications above, and can be used as a building block for differentiable non-convex optimization problems.

Implementing optimization layers can be non-trivial as explicit closed-form solutions typically do not exist. The forward pass needs to call into an optimization problem solver and the backward pass typically *cannot* leverage automatic differentiation. The backwards pass is usually implemented by implicitly differentiating the KKT conditions of the optimization problem as done in bilevel optimization [Gou+16; KP13], sensitivity analysis [Ber99; FI90; BS13], and in our OptNet approach Chapter 3. Thus to implement an optimization layer, users have to manually implement the backwards pass, which is cumbersome and error-prone, or use an existing optimization problem layer such as the differentiable QP layer from Chapter 3, which is not capable of exploiting problem-specific structures, uses dense operations, and requires the user to manually transform their problem into a standard form.

We make `cvxpy` differentiable with respect to the `Parameter` objects provided to the optimization problem by making internal `cvxpy` components differentiable. This involves differentiating the reduction from the `cvxpy` language to the problem data of a cone program in standard form and then differentiating through the cone program. We show how to differentiate through cone programs by implicitly differentiating the residual map from Busseti, Moursi, and Boyd [BMB18], which is of standalone interest as this shows how to differentiate through optimization problems with non-polytope constraints.

5.2 Background

5.2.1 The `cvxpy` modeling language

`cvxpy` [DB16] is a domain-specific modeling language based on disciplined convex programming [GBY06] that allows users to express optimization problems in a more natural way than the standard form required my most optimization problem solvers. `cvxpy` works by transforming the optimization problem from their domain-specific language to a standard (or canonical) form that is passed into a solver. This inner canonicalized problem is then solved and the results are returned to the user. In this chapter, we focus on the canonicalization to a cone program, which is one of the most commonly used modes as most convex optimization problems can be expressed as a cone program, although we note that our method can be applied to other `cvxpy` solvers. Figure 5.1 overviews the relevant `cvxpy` components.

5.2.2 Cone Preliminaries

A set $\mathcal{K}$ is a *cone* if for all $x \in \mathcal{K}$ and $t > 0$, $tx \in \mathcal{K}$. The *dual cone* of a cone $\mathcal{K}$ is

$$\mathcal{K}^* = \left\{ y \;\middle|\; \inf_{x \in \mathcal{K}} y^\top x \geq 0 \right\}.$$

Commonly used cones include the nonnegative orthant $\left\{ x \;\middle|\; x \geq 0 \right\}$, second-order cone $\left\{ (x,t) \in \mathbb{R}^n_+ \;\middle|\; t \geq ||x||_2 \right\}$, positive semidefinite cone $\{ X = X^\top \succeq 0 \}$, and exponential cone

$$\left\{ (x,y,z) \;\middle|\; y > 0, ye^{x/y} \leq z \right\} \cup \left\{ (x,0,z) \;\middle|\; x \leq 0, z \geq 0 \right\} \tag{5.2}$$

We can also create a cone from the Cartesian products of simpler cones as $\mathcal{K} = \mathcal{K}_1 \times \ldots \times \mathcal{K}_p$.

5.2.3 Cone Programming

Most convex optimization problems can be represented and efficiently solved as a cone program that uses the nonnegative orthant, second-order cone, positive semidefinite cone, and exponential cones. This applicability makes them a commonly used internal solver for `cvxpy`, which implements many of the well-known transformations from problems to their conic form. In the following we state properties of cone programs and useful definitions

for this chapter. More details about cone programming can be found in Boyd and Vandenberghe [BV04], Ben-Tal and Nemirovski [BN01], Busseti, Moursi, and Boyd [BMB18], O'Donoghue, Chu, Parikh, and Boyd [ODo+16], Lobo, Vandenberghe, Boyd, and Lebret [Lob+98], and Alizadeh and Goldfarb [AG03].

In their primal (P) and dual (D) forms, cone programs can be represented as

$$\begin{array}{llll} \text{(P)} \;\; x^\star, s^\star = & \operatorname{argmin}_{x,s} & c^\top x & \\ & \text{subject to} & Ax + s = b & \\ & & s \in \mathcal{K} & \end{array} \qquad \begin{array}{llll} \text{(D)} \;\; y^\star = & \operatorname{argmax}_{y} & b^\top y & \\ & \text{subject to} & A^\top y + c = 0 & \\ & & y \in \mathcal{K}^* & \end{array} \tag{5.3}$$

where $x \in \mathbb{R}^n$ is the *primal variable*, $s \in \mathbb{R}^m$ is the *primal slack variable*, $y \in \mathbb{R}^m$ is the *dual variable*. and $\mathcal{K}$ is a nonempty, closed, convex cone with dual cone $\mathcal{K}^*$.

The KKT optimality conditions. The Karush–Kuhn–Tucker (KKT) conditions for the cone program in Equation (5.3) provide necessary and sufficient conditions for optimality and are defined by

$$Ax + s = b, \quad A^\top y + c = 0, \quad s \in \mathcal{K}, \quad y \in \mathcal{K}^*, \quad s^\top y = 0. \tag{5.4}$$

The complimentary slackness condition $s^\top y = 0$ can alternatively be captured with a condition that makes the duality gap zero $c^\top x + b^\top y = 0$.

The homogenous self-dual embedding. Ye, Todd, and Mizuno [YTM94] converts the primal and cone dual programs in Equation (5.3) into a single feasibility problem called the homogenous self-dual embedding, which is defined by

$$Qu = v, \quad u \in \boldsymbol{\mathcal{K}}, \quad v \in \boldsymbol{\mathcal{K}}^*, \quad u_{m+n+1} + v_{m+n+1} > 0, \tag{5.5}$$

where

$$\boldsymbol{\mathcal{K}} = \mathbb{R}^n \times \mathcal{K}^* \times \mathbb{R}_+, \quad \boldsymbol{\mathcal{K}}^* = \{0\}^n \times \mathcal{K} \times \mathbb{R}_+,$$

and Q is the skew-symmetric matrix

$$Q = \begin{bmatrix} 0 & A^\top & c \\ -A & 0 & b \\ -c^\top & -b^\top & 0 \end{bmatrix}.$$

A solution to this embedding problem $(u^\star, v^\star)$ can be used to determine the solution of a conic problem, or to certify the infeasibilty of the problem if a solution doesn't exist. If a solution exists, then $u^\star = (x^\star/\tau, y^\star/\tau, \tau)$ and $v^\star = (0, s^\star/\tau, 0)$.

The conic complementarity set. The *conic complementarity set* is defined by

$$\mathcal{C} = \left\{(u, v) \in \boldsymbol{\mathcal{K}} \times \boldsymbol{\mathcal{K}}^* \;\middle|\; u^\top v = 0\right\}. \tag{5.6}$$

We denote the Euclidean projection onto $\boldsymbol{\mathcal{K}}$ with Π and the Euclidean projection onto $-\boldsymbol{\mathcal{K}}^*$ with Π^*. Moreau [Mor61] shows that $\Pi^* = I - \Pi$.

Minty's parameterization of the complementarity set. Minty's parameterization $M : \mathbb{R}^{m+n+1} \to \mathcal{C}$ of $\mathcal{C}$ is defined by $M(z) = (\Pi z, -\Pi^* z)$. This parameterization is invertible with $M^{-1}(u, v) = u - v$. See Rockafellar [Roc70, Corollary 31.5.1] and Bauschke and

Combettes [BC17, Remark 23.23(i)] for more details. The homogeneous self-dual embedding can be expressed using Minty's parameterization as $-\Pi^* z = Q\Pi z$ where $z_{m+n+1} \neq 0$.

The residual map of Minty's parameterization. Busseti, Moursi, and Boyd [BMB18] defines the *residual map* of Minty's parameterization $\mathcal{R} : \mathbb{R}^{m+n+1} \to \mathbb{R}^{m+n+1}$ as

$$\mathcal{R}(z) = Q\Pi z + \Pi^* z = ((Q - I)\Pi + I)z. \tag{5.7}$$

and shows how to compute the derivative of it when Π is differentiable at z as

$$\mathsf{D}_z\mathcal{R}(z) = (Q - I)\mathsf{D}_z\Pi(z) + I, \tag{5.8}$$

where $z \in \mathbb{R}^{m+n+1}$. The cone projection differentiation $\mathsf{D}_z\Pi(z)$ can be computed as described in Ali, Wong, and Kolter [AWK17].

The Splitting Conic Solver (SCS). SCS [ODo+16] is an efficient way of solving general cone programs by using the alternating direction method of multipliers (ADMM) [Boy+11] and is a commonly used solver with `cvxpy`. In the simplified form, each iteration of SCS consists of the following three steps:

$$\begin{aligned} \tilde{u}^{k+1} &= (I + Q)^{-1}(u^k + v^k) \\ u^{k+1} &= \Pi\left(\tilde{u}^{k+1} - v^k\right) \\ v^{k+1} &= v^k - \tilde{u}^{k+1} + u^{k+1}. \end{aligned} \tag{5.9}$$

The first step projects onto an affine subspace, the second projects onto the cone and the last updates the dual variable. In this paper we will mostly focus on solving the affine subspace projection step. O'Donoghue, Chu, Parikh, and Boyd [ODo+16, Section 4.1] shows that the affine subspace projection can be reduced to solving linear systems of the form

$$\begin{bmatrix} I & -A^\top \\ -A & -I \end{bmatrix} \begin{bmatrix} z_x \\ -z_y \end{bmatrix} = \begin{bmatrix} w_x \\ w_y \end{bmatrix}, \tag{5.10}$$

which can be re-written as

$$z_x = (I + A^\top A)^{-1}(w_x - A^\top w_y), \quad z_y = w_y + Az_x. \tag{5.11}$$

5.3 Differentiating `cvxpy` and Cone Programs

We have created a differentiable `cvxpy` layer by making the relevant components differentiable, which we visually show in Figure 5.1. We make the transformation from the problem data in the original form to the problem data of the canonicalized cone problem differentiable by replacing the numpy operations for this component with PyTorch [Pas+17b] operations. We then pass this data into a differentiable cone program solver, which we show how to create in Section 5.3.1 by implicitly differentiating the residual map of Minty's parameterization for the backward pass. The solution of this cone program can then be mapped back up to the original problem space in a differentiable way and returned.

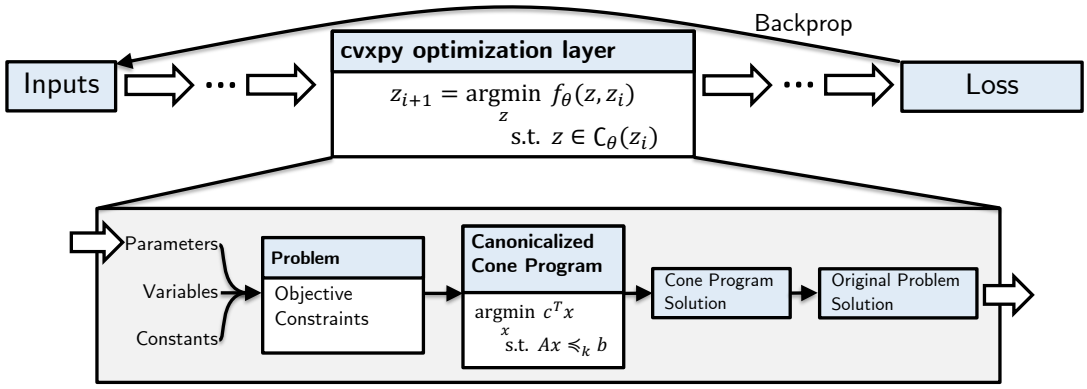


Figure 5.1: Summary of our differentiable `cvxpy` layer that allows users to easily turn most convex optimization problems into layers for end-to-end machine learning.

5.3.1 Differentiating Cone Programs

We consider the argmin of the primal cone program in Equation (5.3) as a function that maps from the problem data $\theta = \{c, A, b\}$ to the solution $x^\star$. The approach from Chapter 3 that differentiates through convex quadratic programs by implicitly differentiating the KKT conditions is difficult to use for cone programs. This is because the cone constraints in the KKT conditions in Equation (5.4) make it difficult to form a set of implicit functions. Instead of implicitly differentiating the KKT conditions, we show how to similarly apply implicit differentiation to the residual map of Minty's parameterization shown in Busseti, Moursi, and Boyd [BMB18] to compute the derivative $\partial x^\star / \partial \theta$. Furthermore for backpropagation, the full Jacobian is expensive and unnecessary to form and we show how to efficiently compute $\partial \ell / \partial \theta$ given $\partial \ell / \partial x^\star$.

Implicit differentiation of the residual map

We assume that we have solved the forward pass of the cone program in Equation (5.3) and have a solution $x^\star, s^\star, y^\star$. We now show how to compute $\partial x^\star / \partial \theta$. This derivation was concurrently considered and done by Agrawal, Barratt, Boyd, Busseti, and Moursi [Agr+19].

We construct $u^\star = (x^\star, y^\star, 1)$, and $v^\star = (0, s^\star, 0)$, and $z^\star = u^\star - v^\star$. The residual map of Minty's parameterization is zero, $R(z^\star) = 0$, and forms a set of implicit equations that describe the solution mapping. Implicit differentiation can be done as described in Dontchev and Rockafellar [DR09] with

$$\mathsf{D}_\theta z^\star = -\left(\mathsf{D}_z \mathcal{R}(z^\star)\right)^{-1} \mathsf{D}_\theta \mathcal{R}(z^\star). \tag{5.12}$$

$\left(\mathsf{D}_z \mathcal{R}(z^\star)\right)^{-1}$ can be computed as described in [BMB18] and $\mathsf{D}_\theta \mathcal{R}(z^\star)$ can be analytically computed. We consider the scaling factor $\tau = z_{m+n+1} = 1$ to be a constant because a solution to the cone program exists. Finally, applying the chain rule to $u^\star = \Pi z^\star$ gives

$$\mathsf{D}_\theta u^\star = (\mathsf{D}_z \Pi z) \mathsf{D}_\theta z^\star. \tag{5.13}$$

We note that implicitly differentiating the residual map captures implicit differentiation of the KKT conditions as a special case for simple cones such as the zero cone and non-negative orthant.

The linear system in Equation (5.12) can be expensive to solve. In special cases such as quadratic programs and LQR problems that we discussed in Chapter 3 and **??**, respectively, this system can be interpreted as the solution to another convex optimization problem and efficiently solved with a method similar to the forward pass. This connection is made by interpreting the linear system solve as a KKT system solve that represents another optimization problem. However for general cone programs it is more difficult to interpret this linear system as a KKT system because of the cone projections and therefore it is more difficult to interpret this linear system solve as an optimization problem.

5.4 Implementation

5.4.1 Forward Pass: Efficiently solving batches of cone programs with SCS and PyTorch

Naïvely implemented optimization layers can become computational bottlenecks when used in a machine learning pipeline that requires efficiently processing minibatches of data. This is because most other parts of the modeling pipeline involve operations such as linear maps, convolutions, and activation functions that can quickly be executed on the GPU to exploit data parallelism across the minibatch. Most off-the-shelf optimization problem solvers are designed for the setting of solving a single problem at a time and are not easily able to be plugged into the batched setting required when using optimization layers.

To overcome the computational challenges of solving batches of cone programs concurrently, we have created a batched PyTorch and potentially GPU-backed backend for the Splitting Conic Solver (SCS) [ODo+16]. The bottleneck of the SCS iterates in Equation (5.9) is typically in the subspace projection part that solves linear systems of the form

$$\tilde{u}^{k+1} = (I + Q)^{-1}(u^k + v^k) \tag{5.14}$$

We have added a new linear system solver backend to the official SCS C implementation that calls back up into Python to solve this linear system.

Our cone program layer implementation offers the following modes for solving a batch of N cone programs represented in standard form as in Equation (5.3) with as $\theta_i = \{A_i, b_i, c_i\}$ for $i \in \{1, \ldots, N\}$ with SCS. As common in practice, we assume that the cone programs have the structure and use the same cones but have different problem data θ_i. We empirically compare these modes in Section 5.6.

Vanilla SCS, serialized. This is a baseline mode that is the easiest to implement and sequentially iterates through the problems θ_i. This lets us use the vanilla SCS sparse direct and indirect linear system solvers on the CPU and CUDA, but does not take advantage of data parallelism.

Vanilla SCS, batched. This is another baseline mode that comes from observing that a batch of cone programs can be represented as a single cone program in standard form as in Equation (5.3) with variables $x = [x_1^\top, \ldots, x_N^\top]^\top$ and data $A = \mathrm{diag}(A_1, \ldots, A_N)$, $b = [b_1^\top, \ldots, b_N^\top]^\top$, and $c = [c_1^\top, \ldots, c_N^\top]^\top$. This exploits the knowledge that all of the cone programs can be solved concurrently. The bottleneck of this mode is still typically in the linear system solve portion of SCS, which happens using sparse operations on the CPU or GPU.

SCS+PyTorch, batched. In this mode we represent the batch of cone programs as a single batched cone program use SCS will callbacks up into Python so that we can use PyTorch to efficiently solve the linear system. This allows us to keep the A data in PyTorch and potentially on the GPU without converting/transferring it and passing it into the SCS. Specifically we use dense operations and have implemented direct and indirect methods to solve Equation (5.11) in PyTorch and then pass the result back down into SCS for the rest of the operations. Our direct method uses PyTorch's batched LU factorizations and solves and our indirect method uses a batched conjugate gradient (CG) implementation. These custom linear system solvers are able to explicitly take advantage of the independence present in the linear systems that the sparse linear system solvers may not recognize automatically, and the dense solvers are also useful for dense cone programs, which come up in the context of differentiable optimization layers when large portions of the constraints are being learned.

5.4.2 Backward pass: Efficiently solving the linear system

When using cone programs as layers in end-to-end learning systems with some scalar-valued loss function ℓ, the full Jacobian $\mathsf{D}_\theta x^\star$ is expensive and unnecessary to form and requires solving $|\theta|$ linear systems. The Jacobian is only used when applying the chain rule to obtain $\mathsf{D}_\theta \ell = (\mathsf{D}_{x^\star}\ell)\mathsf{D}_\theta x^\star$. We can directly compute $\mathsf{D}_\theta \ell$ without computing the intermediate Jacobian by solving a single linear system. Following the method of Chapter 3, we set up the system

$$\mathsf{D}_z \mathcal{R}(z^\star) \begin{bmatrix} d_{z_1} \\ d_{z_2} \\ 0 \end{bmatrix} = - \begin{bmatrix} \nabla_{x^\star}\ell \\ 0 \\ 0 \end{bmatrix}. \tag{5.15}$$

Applying the chain rule to $u^\star = \Pi z^\star$ gives $d_x = d_{z_1}$ and $d_y = (\mathsf{D}_z \Pi z) d_{z_2}$. We then compute the relevant backpropagation derivatives as

$$\nabla_c \ell = d_x \qquad \nabla_A \ell = d_y \otimes x^\star + y^\star \otimes d_x \qquad \nabla_b \ell = -d_y \tag{5.16}$$

Solving Equation (5.15) is still challenging to implement in practice as $\mathsf{D}_z \mathcal{R}(z^\star)$ can be large and sparse and doesn't have obviously exploitable properties such as symmetry or anti-symmetry. In addition to directly solving this linear system, we also explore the use of LSQR [PS82] as an iterative indirect method of solving this system in Section 5.6.2. Our LSQR implementation uses the implementation from Ali, Wong, and Kolter [AWK17] to compute $\mathsf{D}_z \Pi(z)$ in the form of an abstract linear operator so the full matrix does not need to be explicitly formed.

5.5 Examples

This section provides example use cases of our `cvxpy` optimization layer. All of these use the preamble

```
import cvxpy as cp
from cvxpyth import CvxpyLayer
```

5.5.1 The ReLU, sigmoid, and softmax

We will start with basic examples and revisit the optimization views of the ReLU, sigmoid, and softmax from Section 2.4.4. These can be implemented with our `cvxpy` layer in a few lines of code.

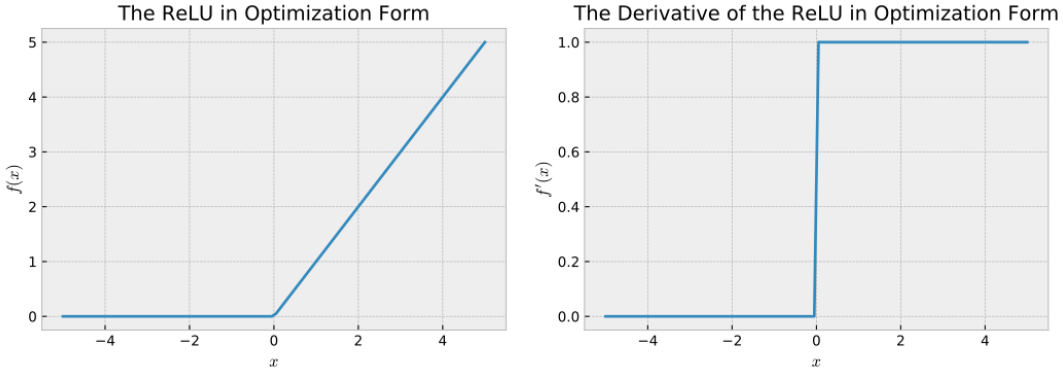
The ReLU. Recall from Equation (2.2) that the optimization view is

$$f(x) = \underset{y}{\operatorname{argmin}} \quad \frac{1}{2}||x-y||_2^2 \quad \text{s.t.} \quad y \geq 0.$$

We can implement this layer with:

```
x = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(cp.sum_squares(y-x))
cons = [y >= 0]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[x], out_vars=[y])
```

This layer can be used and differentiated through just as any other PyTorch layer. Here is the output and derivative for a single dimension, illustrating that this is indeed performing the same operation as the ReLU.



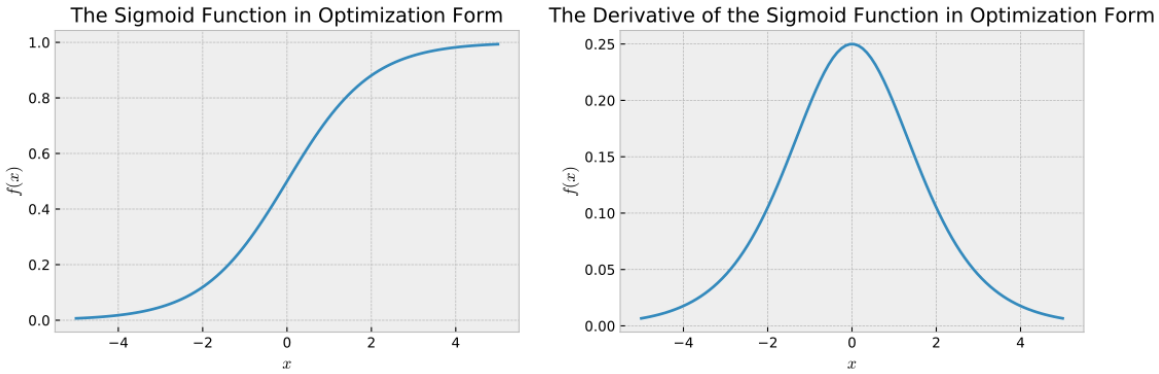
The sigmoid. Recall from Equation (2.4) that the optimization view is

$$f(x) = \underset{0<y<1}{\operatorname{argmin}} \quad -x^\top y - H_b(y).$$

We can implement this layer with:

```
x = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(-x.T*y - cp.sum(cp.entr(y) + cp.entr(1.-y)))
prob = cp.Problem(obj)
layer = CvxpyLayer(prob, params=[x], out_vars=[y])
```

We can also check that the output and derivative matches what we expect from the usual sigmoid function:



The softmax. Lastly recall from Equation (2.5) that the optimization view is

$$f(x) = \underset{0<y<1}{\operatorname{argmin}} \quad -x^\top y - H(y) \quad \text{s.t.} \quad 1^\top y = 1$$

We can implement this layer with:

```
x = cp.Parameter(d)
y = cp.Variable(d)
obj = cp.Minimize(-x.T*y - cp.sum(cp.entr(y)))
cons = [sum(y) == 1.]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[x], out_vars=[y])
```

5.5.2 The OptNet QP

We can re-implement the OptNet QP layer from Chapter 3 with our differentiable `cvxpy` layer in a few lines of code. The OptNet layer is represented as a convex quadratic program of the form

$$\begin{aligned} x^\star = \underset{x}{\text{argmin}} \;\; & \frac{1}{2} x^\top Q x + p^\top x \\ \text{subject to} \;\; & Ax = b \\ & Gx \leq h \end{aligned} \tag{5.17}$$

where $x \in \mathbb{R}^n$ is our optimization variable $Q \in \mathbb{R}^{n \times n} \succeq 0$ (a positive semidefinite matrix), $p \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $G \in \mathbb{R}^{p \times n}$ and $h \in \mathbb{R}^p$ are problem data. We can implement this with:

```
Q = cp.Parameter((n, n), PSD=True)
p = cp.Parameter(n)
A = cp.Parameter((m, n))
b = cp.Parameter(m)
G = cp.Parameter((p, n))
h = cp.Parameter(p)
x = cp.Variable(n)
obj = cp.Minimize(0.5*cp.quad_form(x, Q) + p.T * x)
cons = [A*x == b, G*x <= h]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[Q, p, A, b, G, h], out=[x])
```

This layer can then be used by passing in the relevant parameter values:

```
Lval = torch.randn(nx, nx, requires_grad=True)
Qval = Lval.t().mm(Lval)
pval = torch.randn(nx, requires_grad=True)
Aval = torch.randn(ncon_eq, nx, requires_grad=True)
bval = torch.randn(ncon_eq, requires_grad=True)
Gval = torch.randn(ncon_ineq, nx, requires_grad=True)
hval = torch.randn(ncon_ineq, requires_grad=True)
y = layer(Qval, pval, Aval, bval, Gval, hval)
```

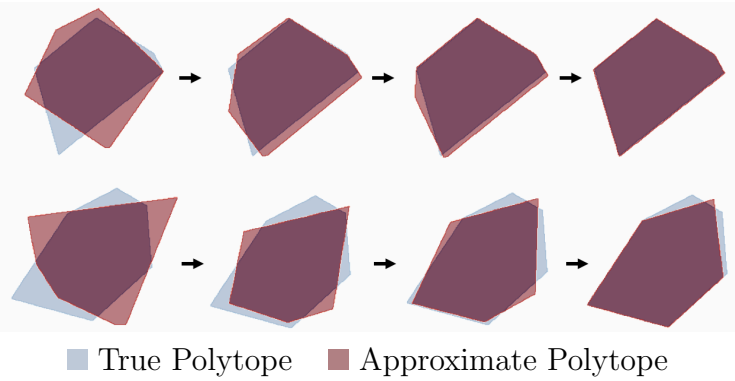


Figure 5.2: Learning polyhedrally constrained problems.

5.5.3 Learning Polyhedral Constraints

We demonstrate how gradient-based learning can be done with a `cvxpy` layer in this synthetic example. Consider the polyhedrally constrained projection problem

$$\begin{aligned} \hat{y} = \underset{y}{\text{argmin}} \quad & \frac{1}{2}||p-y||_2^2 \\ \text{s.t.} \quad & Gy \leq h \end{aligned}$$

Suppose we don't know the polytope's parameters $\theta = \{G, h\}$ and want to learn them from data. Then using the MSE for ℓ, we can randomly initialize ellipsoids θ and learn them with gradient steps $\nabla_\theta \ell$. We note that this problem is meant for illustrative purposes and could be solved by taking the convex hull of the input data points. However our approach would still work if this was over a latent and unobserved part of the model, of if you want to take an approximate convex hull that limits the number of polytope edges.

We can implement this layer with the following code. Figure 5.2 shows the results of learning on two examples. Each problem has a true known polytope that we show in blue and the model's approximation is in red. Learning starts on the left with randomly initialized polytopes that are updated with gradient steps, which are shown in the images progressing to the right.

```
G = cp.Parameter((m, n))
h = cp.Parameter(m)
p = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(0.5*cp.sum_squares(y-p))
cons = [G*y <= h]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[p, G, h], out=[y])
```

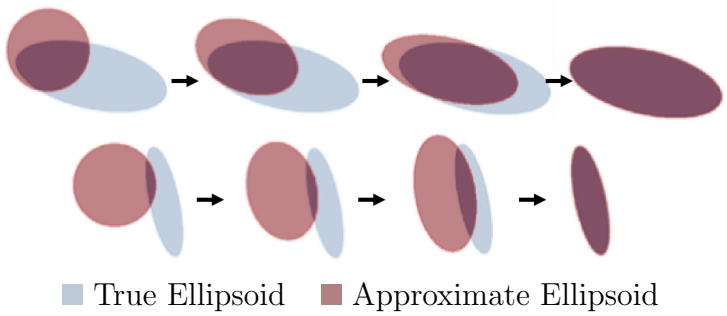


Figure 5.3: Learning ellipsoidally constrained problems.

5.5.4 Learning Ellipsoidal Constraints

In addition to learning polyhedral constraints, we can easily learn any parameterized convex constraint set. Suppose instead that we want to learn an ellipsoidal projection of the form

$$\begin{aligned} \hat{y} = \underset{y}{\text{argmin}} \quad & \frac{1}{2}||p-y||_2^2 \\ \text{s.t.} \quad & \frac{1}{2}(y-z)^\top A(y-z) \leq 1 \end{aligned}$$

with ellipsoid parameters $\theta = \{A, z\}$ This is an interesting optimization problem to consider because it is an example of doing learning with a non-polytope cone program (a SOCP), which prior approaches such as OptNet could not easily handle.

We can implement this layer with the following code. Figure 5.3 visualizes the learning process on two examples. As before, each problem has a true known ellipsoid that we show in blue and the model's approximation is in red. Learning starts on the left with randomly initialized ellipsoids that are updated with gradient steps, which are shown in the images progressing to the right.

```
A = cp.Parameter((n, n), PSD=True)
z = cp.Parameter(n)
p = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(0.5*cp.sum_squares(y-p))
cons = [0.5*cp.quad_form(y-z, A) <= 1]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[p, A, z], out=[y])
```

5.6 Evaluation

In this section we analyze the runtime of our layer's forward and backward passes compared to hand-written implementations for commonly used optimization layers. We will focus on three tasks:

Task 1: Dense QP. We consider a QP layer of the form Equation (5.17) with a dense quadratic objective and dense inequality constraints. Our default experiment uses a QP with 100 latent variables, 100 inequality constraints, and a minibatch size of 128 examples. We chose this task to understand how the performance of our `cvxpy` layer compares to the `qpth` implementation from Chapter 3, which we use as a comparison point. The problem size we consider here is comparable to the QP problem sizes considered in Chapter 3. The backwards pass of `qpth` is optimized to use a single batched, pre-factorized linear system solve.

Task 2: Box QP. We consider a QP layer of the form Equation (5.17) with a dense quadratic objective constrained to the box $[-1, 1]^n$. Our default experiment uses a QP with 100 latent variables and a minibatch size of 128 examples. We chose this task to study the impacts of sparsity on the runtime. We again use `qpth` as the comparison point for these experiments.

Task 3: Linear Quadratic Regulator (LQR). We consider a continuous-state-action, discrete-time, finite-horizon LQR problem of the form

$$\tau_{1:T}^\star = \underset{\tau_{1:T}}{\text{argmin}} \sum_t \frac{1}{2}\tau_t^\top C_t \tau_t + c_t^\top \tau_t \quad \text{subject to} \quad x_1 = x_{\text{init}},\ x_{t+1} = F_t\tau_t + f_t. \tag{5.18}$$

where $\tau_{1:T} = \{x_t, u_t\}_{1:T}$ is the nominal trajectory, T is the horizon, x_t, u_t are the state and control at time t, $\{C_t, c_t\}$ parameterize a convex quadratic cost, and $\{F_t, f_t\}$ parameterize an affine system transition dynamics. We consider a control problem with 10 states, 2 actions, and a horizon of 5. We compare to the differentiable model predictive control (MPC) solver from [Amo+18b], which uses batched PyTorch operations to solve a batch of LQR problems with the Riccati equations, and then implements the backward pass with another, simpler, LQR solve with the Riccati equations.

For each of these tasks we have measured the forward and backward pass execution times for our layer in comparison to the specialized solvers. We have run these experiments on an unloaded system with an NVIDIA GeForce GTX 1080 Ti GPU and a four-core 2.60GHz Intel Xeon E5-2623 CPUs hyper-threaded to eight cores. We set the number of OpenMP threads to 8 for our experiments. For numerical stability, we use 64-bits for all of our implementations and baselines. For `qpth` and our implementation, we use an iteration stopping condition of $\epsilon = 10^{-3}$.

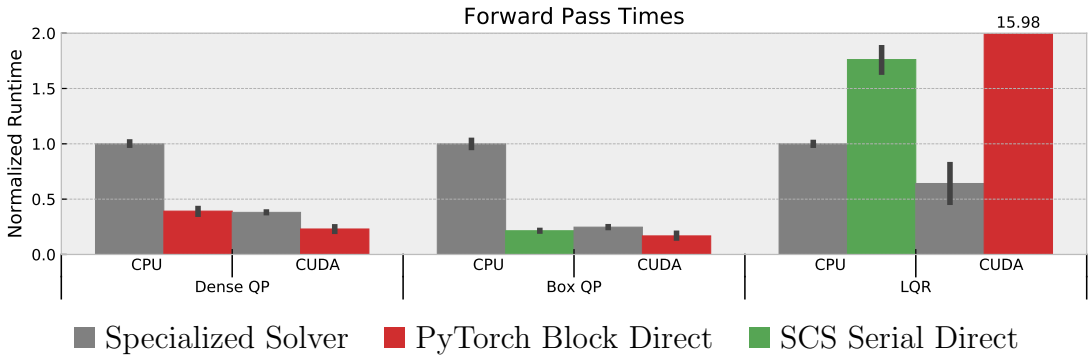


Figure 5.4: Forward pass execution times. For each task we run ten trials on an unloaded system and normalize the runtimes to the CPU execution time of the specialized solver. The bars show the 95% confidence interval. For our method, we show the best performing mode.

5.6.1 Forward pass profiling

Figure 5.4 summarizes our main forward pass execution times. Figure 5.6 shows the runtimes of all of the modes and batch sizes, and Figure 5.5 illustrates the speedup of our best mode compared to the specialized solvers. We have implemented and run every mode from Section 5.4.1 and our summary presents the best-performing mode, which in every case on the GPU is our block direct solver. On the CPU, serializing SCS calls is competitive for problems with more sparsity. For dense and sparse QPs on the CPU and GPU, our batched SCS+PyTorch direct cone solver is faster than the `qpth` solver, which likely comes from the acceleration, convergence, and normalization tricks in SCS that are not present in `qpth`. The LQR task presents a sparse problem that illustrates the challenges to using a general cone program formulation. Our specialized solver that solves the Riccati equations in batched form exploits the sparsity pattern of the problem that is extremely difficult for the general cone program formulation we consider here to take advantage of. If the correct mappings to the cone program exist, our layer could be modified to accept an optimized user-provided solver for the forward pass so that users can still take advantage of our backward pass implementation.

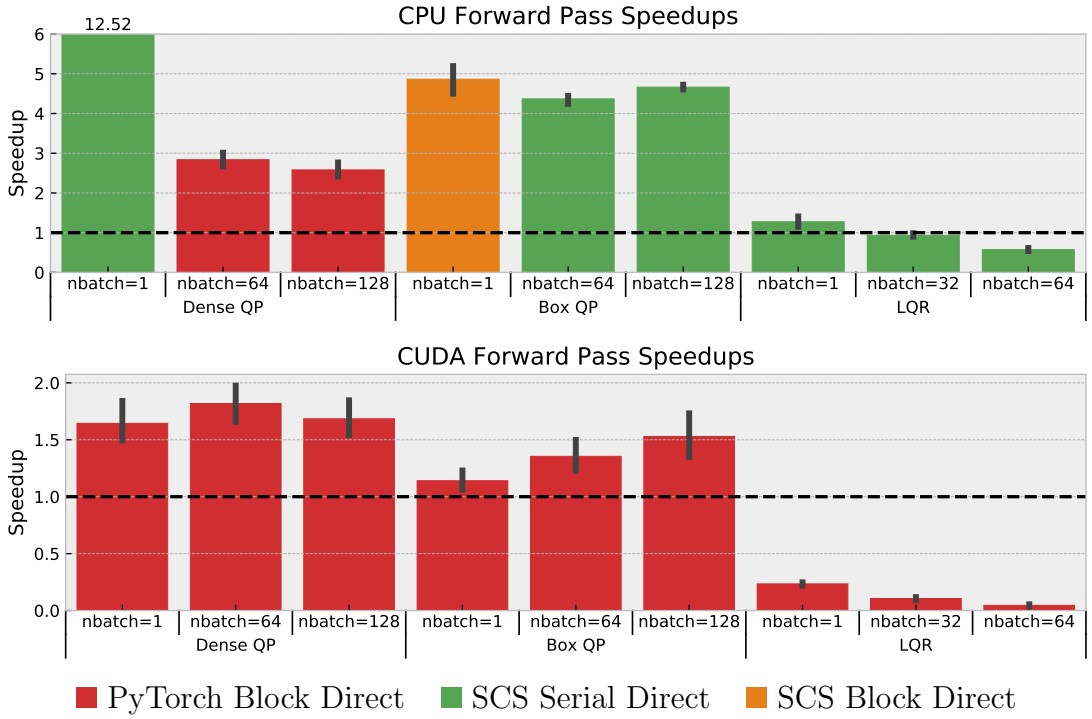


Figure 5.5: Forward pass execution time speedups of our best performing method in comparison to the specialized solver's execution time. For each task we run ten trials on an unloaded system. The bars show the 95% confidence interval.

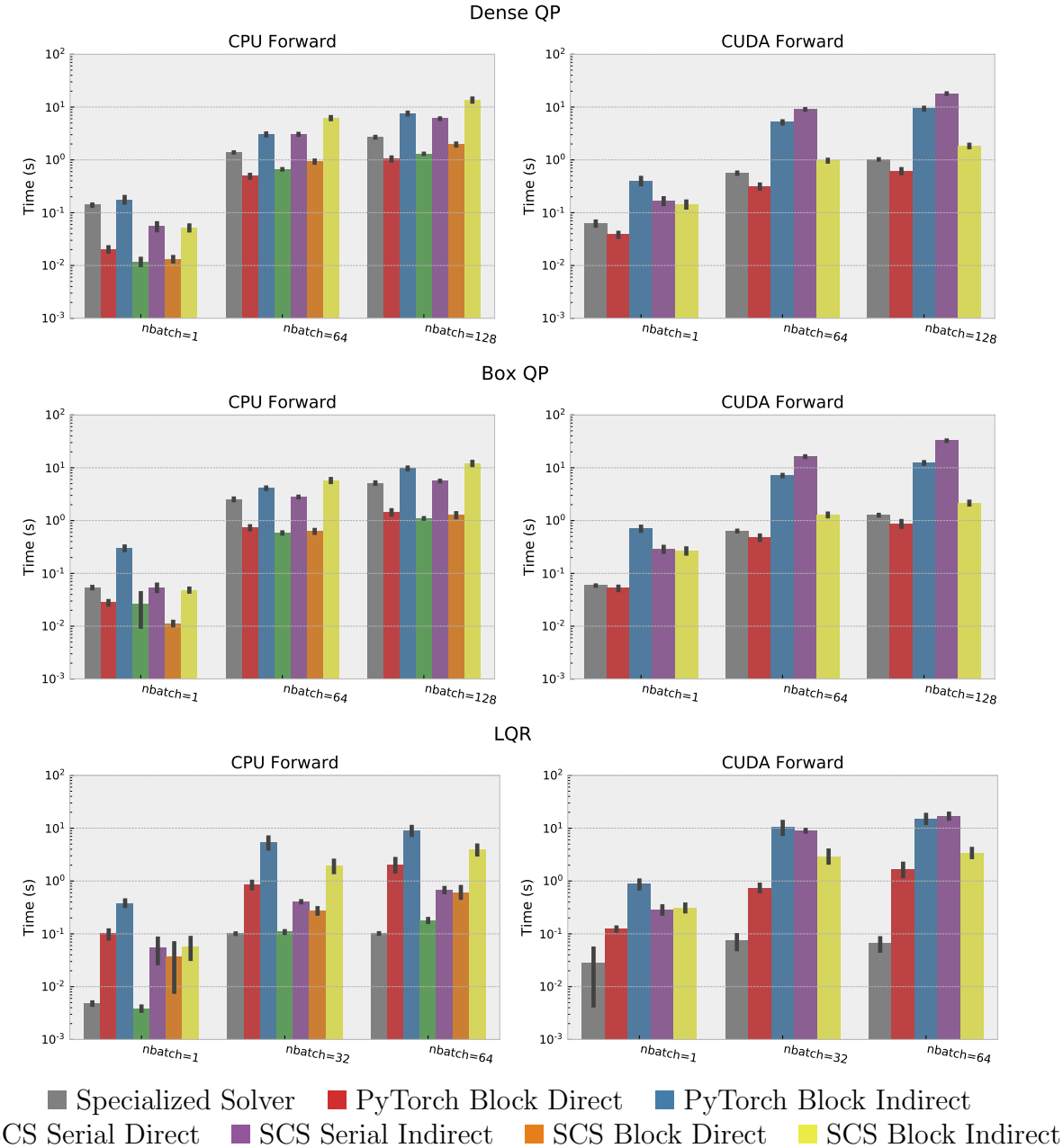


Figure 5.6: Full data for the forward pass execution times. For each task we run ten trials on an unloaded system. The bars show the 95% confidence interval.

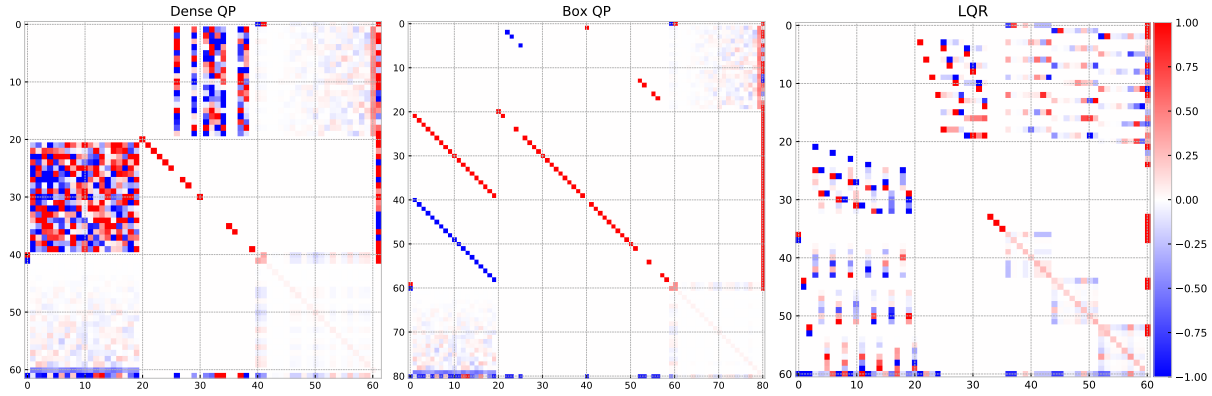


Figure 5.7: Sample linear system coefficients for the backward pass system in Equation (5.15) on smaller versions of the tasks we consider. The tasks we consider are approximately five times larger than these systems.

5.6.2 Backward pass profiling

In this section we compare the backward pass times of our layer in comparison to the specialized solvers on the same three tasks as before: the dense QP, the box QP, and LQR. We show that differentiating through our conic solver is competitive in comparison to the specialized solver. As a comparison point, the `qpth` solver exploits the property that the linear system for the backward pass is the same as the linear system in the forward pass and can therefore do the backward pass with a single pre-factorized solve. The LQR solver exploits the property that the backward pass for LQR can be interpreted as another LQR problem that can efficiently be solved with the Riccati recursion.

These comparisons are important because the linear system for differentiating cone programs in Equation (5.15) is a more general form and cannot leverage the same exploits as the specialized solvers. To get an intuition of what these linear systems look like on our tasks, we plot sample maps on smaller problems of the coefficient matrix in Figure 5.7. This illustrates the sparsity that is typically present in the linear system that needs to be solved, but also illustrates that beyond sparsity, there is no other common property that can be exploited between the tasks.

To understand how many LSQR iterations are necessary to solve our task, we compare the approximate derivatives computed by LSQR to the derivatives obtained by directly solving the linear system in Figure 5.8. This shows that typically 500-1000 LSQR iterations are necessary for the tasks that we consider. In some cases such as $\partial x^\star / \partial A$ for LQR, the approximate gradient computed by LSQR does never converges exactly to the true gradient.

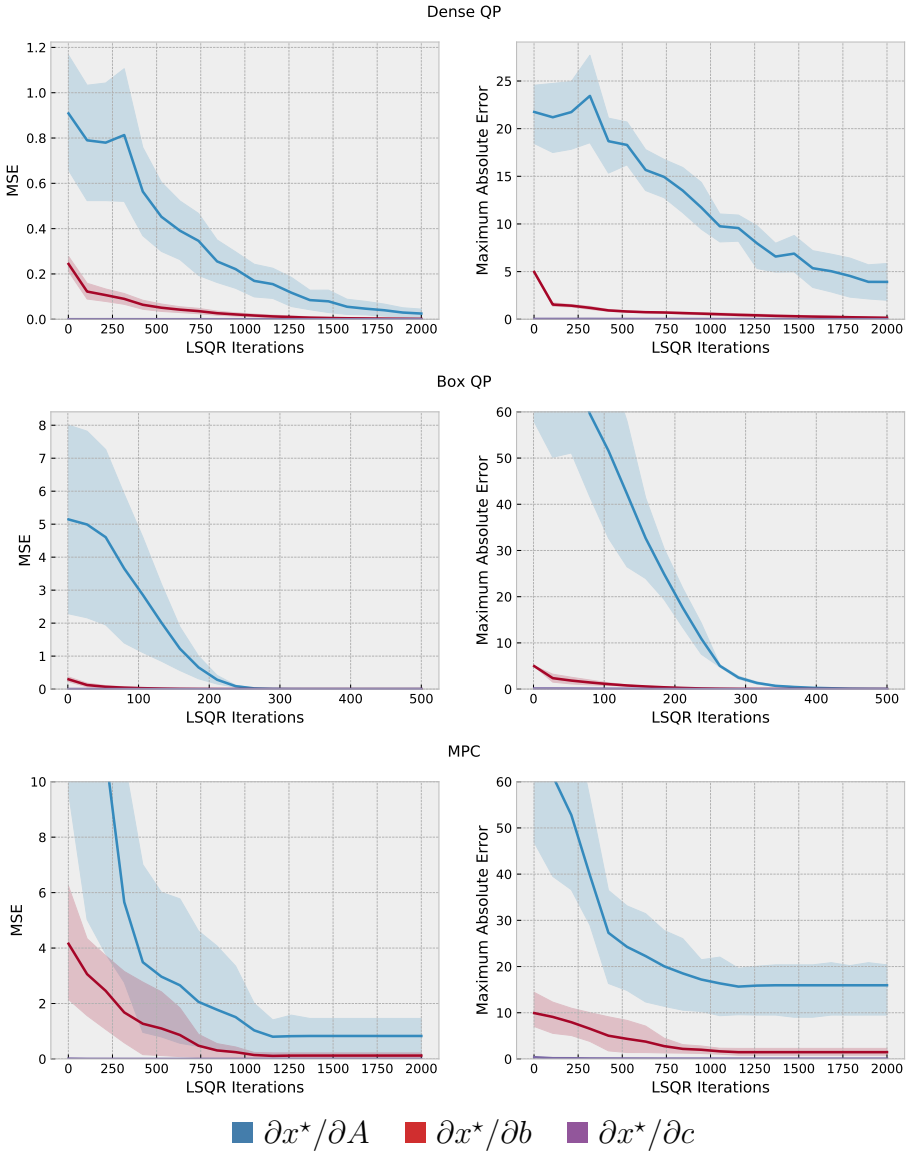


Figure 5.8: LSQR convergence for the backward pass systems. The shaded areas show the 95% confidence interval across ten problem instances.

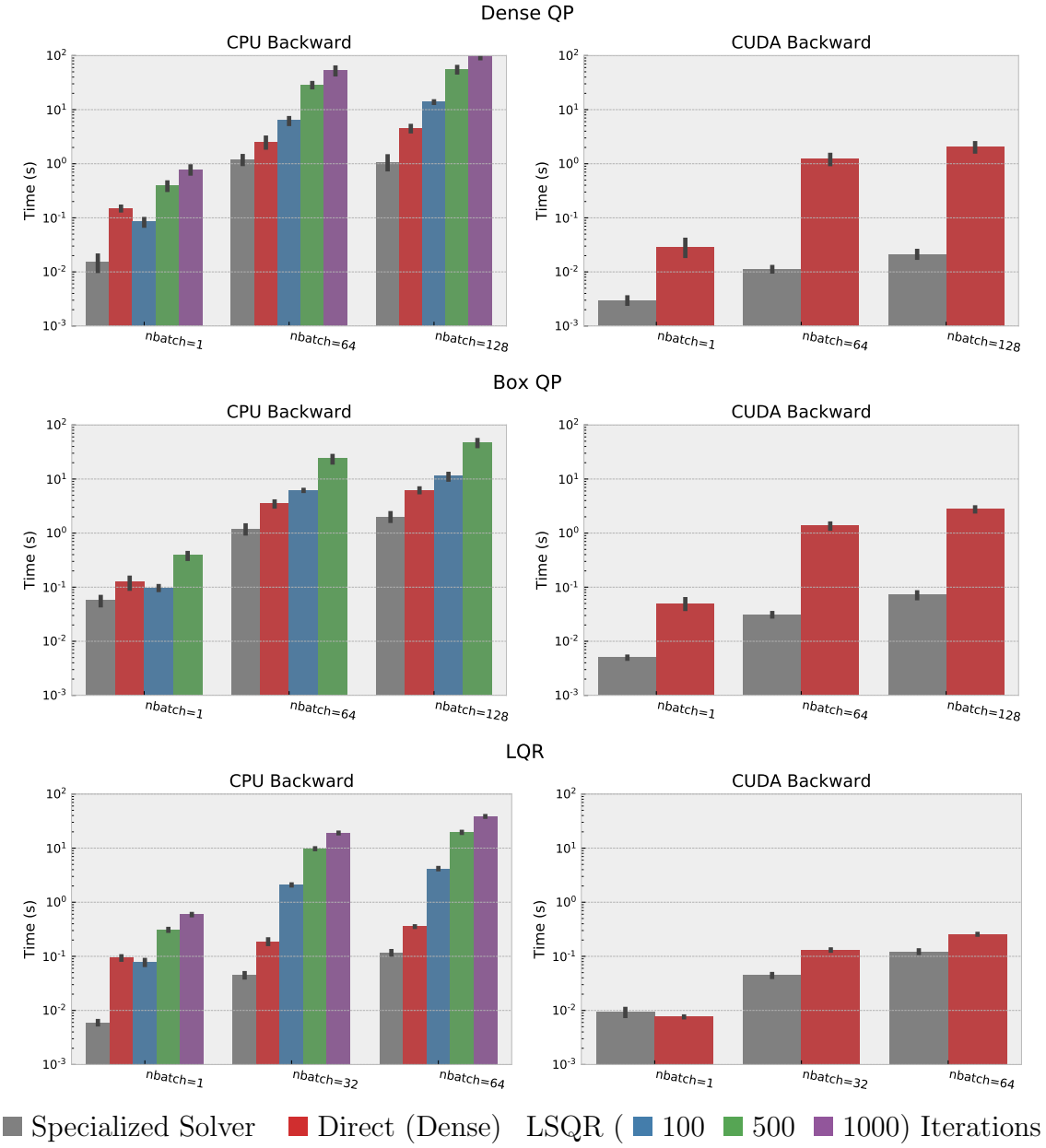


Figure 5.9: Backward pass execution times. For each task we run ten trials on an unloaded system. The bars show the 95% confidence interval.

Figure 5.9 compares our backward pass times to the specialized solvers for the QP and LQR tasks. This shows that there is a slight computational overhead in comparison to the specialized solvers, but that solving the linear system is still tractable for these tasks. The LSQR runtime is serialized across the batch, and is currently only implemented on the CPU. We emphasize that if the backward pass time becomes a bottleneck, the runtime can be further improved by further exploiting the sparsity by investigating other direct and indirect solvers for the systems, or by exploiting the property that we mentioned earlier in Section 5.3.1 that for simple cones like the free and non-negative cones, parts of the system become the same as parts of the KKT system.

5.7 Conclusion

This section has presented a way of differentiating through cone programs that enabled us to create a powerful prototyping tool for differentiable convex optimization layers. Practitioners can use this library in place of hand-implementing a solver and implicitly differentiating the KKT conditions. The speed of our tool is competitive with the speed of specialized solvers, even in the batched setting required for machine learning.

Part III

Conclusions and Future Directions

Chapter 6

Conclusions and Future Directions

In this thesis we have introduced new building blocks and fundamental components for machine learning that enable optimization-based domain knowledge to be injected into the modeling pipeline. We have presented the *OptNet* architecture as a foundation for convex optimization layers and the *input-convex neural network* architecture as a foundation for deep convex energy-based learning. We have shown how the OptNet approach can be applied to differentiable model-predictive control and top-k learning. To enable rapid prototyping in this space, we have shown how `cvxpy` can be turned into a differentiable layer. Differentiable optimization provides an expressive set of operations and have a promising set of future directions. In the following we provide a brief outlook of how optimization-based modeling benefits seven application areas, and we highlight a few key references in this space.

1. **Game theory.** The game theory literature typically focuses on finding optimal strategies of playing games with known rules. While the rules of a lot of games are known explicitly, scenarios could come up where it's useful to learn the rules of a game and to have a "game theory" equilibrium-finding layer. For example in reinforcement learning, an agent can have an arbitrary differentiable "game theory" layer that is able of representing complex tasks, state spaces, and problems as an equilibrium-finding problem in a game where the rules are automatically extracted. This is explored in Ling, Fang, and Kolter [LFK18].
2. **Stochastic optimization and end-to-end learning.** Typically probabilistic models are used in the context of larger systems. When these systems have an objective that is being optimized, it is usually ideal to incorporate the knowledge of this objective into the probabilistic modeling component. If the downstream systems involve solving stochastic optimization problems, as in power-systems, creating an end-to-end differentiable architecture is more difficult and can be done by using differentiable optimization as in Donti, Amos, and Kolter [DAK17].

3. **Reinforcement learning and control.**
    - **Safety.** RL agents may be deployed in scenarios when the agent should avoid parts of the state space, *e.g.* in safety-critical environments. Differentiable optimization layers can be used to help constrain the policy class so that these regions are avoided. This is explored in Dalal, Dvijotham, Vecerik, Hester, Paduraru, and Tassa [Dal+18] and Pham, De Magistris, and Tachibana [PDT18].
    - **Differentiable control and planning.** The differentiable MPC approach we presented in **??** is just one step towards a significantly broader vision of integrating control and learning for doing imitation or policy learning.
    - **Physics-based modeling.** When RL environments involve physical systems, it may be useful to have a physics-based modeling. This can be done with a differentiable physics engines as in Avila Belbute-Peres, Smith, Allen, Tenenbaum, and Kolter [Avi+18].
    - **Inverse cost and reward learning.** Given observed trajectories in an imitation learning setup, modeling agents as controllers that are optimizing an objective is a powerful paradigm [NR+00; FLA16]. Differentiable controllers are useful when trying to reconstruct an optimization problem that other agents are solving. This is done in the context of cost shaping in Tamar, Thomas, Zhang, Levine, and Abbeel [Tam+17].
    - **Multi-agent environments.** In multi-agent environments, other agents can be modeled as entities that are solving control optimization or other learning problems. This knowledge can be integrated into the learning procedure as in Foerster, Chen, Al-Shedivat, Whiteson, Abbeel, and Mordatch [Foe+18].
    - **Control in high-dimensional state spaces.** Control in high-dimensional state spaces such as visual spaces is challenging and it is typically useful to extract a lower-dimensional latent space from the original feature space. This is typically done by either hand-engineering a feature extractor, or by learning an embedding with an unsupervised learning method as in Watter, Springenberg, Boedecker, and Riedmiller [Wat+15] and Kurutach, Tamar, Yang, Russell, and Abbeel [Kur+18]. Viewing controllers as differentiable entities is reasonable for embedding states because the cost function of the controller can be parameterized to learn a cost associated with the latent representation.
4. **Discrete, combinatorial, and submodular optimization.** The space of discrete, combinatorial, and mixed optimization problems captures an even more expressive set of operations than the continuous and convex optimization problems we have considered in this thesis. Similar optimization components can be made for some of these types of problems and is explored in Djolonga and Krause [DK17], Tschiatschek, Sahin, and Krause [TSK18], Mensch and Blondel [MB18], Niculae, Martins, Blondel, and Cardie [Nic+18], and Niculae and Blondel [NB17].

5. **Meta-learning.** Some meta-learning formulations such as Finn, Abbeel, and Levine [FAL17] and Ravi and Larochelle [RL16] involve learning through an unrolled optimizer that typically solve an unconstrained, continuous, and non-convex optimization problem. In some cases, unrolling through a solver with many iterations may require inefficient amounts of compute or memory. Meta-learning methods can be improved by using differentiable closed-form solvers, as done in MetaOptNet [Lee+19] with a differentiable SVM layer and in Bertinetto, Henriques, Torr, and Vedaldi [Ber+18] with differentiable ridge and logistic regression.
6. **Optimization viewpoints of standard components.** A motivation behind this thesis work has been the optimization viewpoints of standard layers we discussed in Section 2.4.4. Many other directions can be taken with the viewpoints, such as the proximal operator viewpoint in Bibi, Ghanem, Koltun, and Ranftl [Bib+18] that interprets deep layers as stochastic solvers.
7. **Hyper-parameter and generalization optimization.** The learning procedure for many linear machine learning models can be interpreted as the solution to a convex optimization problem over the loss function. This convex learning process can therefore also be made differentiable and used for optimizing the hyper-parameters of these algorithms. This is done in Barratt and Sharma [BS18] to optimize the cross-validation risk of convex formulations, including logistic regression, SVMs, and elastic net regression; for least squares in Barratt and Boyd [BB19]; and SVMs in Lee, Maji, Ravichandran, and Soatto [Lee+19].

Bibliography

This bibliography contains 229 references.

[Aba+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[AG03] Farid Alizadeh and Donald Goldfarb. "Second-order cone programming". In: *Mathematical programming* 95.1 (2003), pp. 3–51.

[Agr+19] Akshay Agrawal, Shane Barratt, Stephen Boyd, Enzo Busseti, and Walaa M. Moursi. "Differentiating Through a Conic Program". In: *arXiv e-prints*, arXiv:1904.09043 (Apr. 2019), arXiv:1904.09043. arXiv: 1904.09043 [math.OC].

[AK17] Brandon Amos and J. Zico Kolter. "OptNet: Differentiable Optimization as a Layer in Neural Networks". In: *Proceedings of the International Conference on Machine Learning*. 2017.

[ALS16] Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. *OpenFace: A general-purpose face recognition library with mobile applications*. Tech. rep. Technical Report CMU-CS-16-118, CMU School of Computer Science, 2016.

[Amo+18a] Brandon Amos, Laurent Dinh, Serkan Cabi, Thomas Rothörl, Sergio Gómez Colmenarejo, Alistair Muldal, Tom Erez, Yuval Tassa, Nando de Freitas, and Misha Denil. "Learning Awareness Models". In: *International Conference on Learning Representations*. 2018.

[Amo+18b] Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J. Zico Kolter. "Differentiable MPC for End-to-end Planning and Control". In: *Advances in Neural Information Processing Systems*. 2018, pp. 8299–8310.

[AQN06] Pieter Abbeel, Morgan Quigley, and Andrew Y Ng. "Using inaccurate models in reinforcement learning". In: *Proceedings of the 23rd international conference on Machine learning*. ACM. 2006, pp. 1–8.

[Avi+18] 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*. 2018, pp. 7178–7189.

[AWK17] Alnur Ali, Eric Wong, and J. Zico Kolter. "A semismooth Newton method for fast, generic convex programming". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 70–79.

[AXK17] Brandon Amos, Lei Xu, and J. Zico Kolter. "Input Convex Neural Networks". In: *Proceedings of the International Conference on Machine Learning*. 2017.

[Bal+16] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. "Deepcoder: Learning to write programs". In: *arXiv preprint arXiv:1611.01989* (2016).

[Ban+17] Somil Bansal, Roberto Calandra, Sergey Levine, and Claire Tomlin. "MBMF: Model-Based Priors for Model-Free Reinforcement Learning". In: *arXiv preprint arXiv:1709.03153* (2017).

[Bar18] Shane Barratt. "On the differentiability of the solution to convex optimization problems". In: *arXiv preprint arXiv:1804.05098* (2018).

[Bat+16] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. "Interaction networks for learning about objects, relations and physics". In: *Advances in neural information processing systems*. 2016, pp. 4502–4510.

[Bat+18] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. "Relational inductive biases, deep learning, and graph networks". In: *arXiv preprint arXiv:1806.01261* (2018).

[BB08] Bradley M Bell and James V Burke. "Algorithmic differentiation of implicit functions and optimal values". In: *Advances in Automatic Differentiation*. Springer, 2008, pp. 67–77.

[BB19] Shane Barratt and Stephen Boyd. "Least Squares Auto-Tuning". In: *arXiv preprint arXiv:1904.05460* (2019).

[BB88] Jonathan Barzilai and Jonathan M Borwein. "Two-point step size gradient methods". In: *IMA Journal of Numerical Analysis* 8.1 (1988), pp. 141–148.

[BC17] Heinz H Bauschke and Patrick L Combettes. "Convex Analysis and Monotone Operator Theory in Hilbert Spaces, second edition". In: (2017).

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[Bel17] David Belanger. "Deep Energy-Based Models for Structured Prediction". PhD thesis. University of Massachusetts Amherst, 2017.

[Ber+05] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Dynamic programming and optimal control*. Vol. 1. 3. Athena scientific Belmont, MA, 2005.

[Ber+18] Luca Bertinetto, João F Henriques, Philip HS Torr, and Andrea Vedaldi. "Meta-learning with differentiable closed-form solvers". In: *arXiv preprint arXiv:1805.08136* (2018).

[Ber82] Dimitri P Bertsekas. "Projected Newton methods for optimization problems with simple constraints". In: *SIAM Journal on control and Optimization* 20.2 (1982), pp. 221–246.

[Ber99] Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

[Bib+18] Adel Bibi, Bernard Ghanem, Vladlen Koltun, and René Ranftl. "Deep Layers as Stochastic Solvers". In: (2018).

[Bis07] Christopher Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics), 1st edn. 2006. corr. 2nd printing edn.* Springer, New York, 2007.

[BLH94] Yoshua Bengio, Yann LeCun, and Donnie Henderson. "Globally trained handwritten word recognizer using spatial representation, convolutional neural networks, and hidden Markov models". In: *Advances in neural information processing systems* (1994), pp. 937–937.

[BM16] David Belanger and Andrew McCallum. "Structured prediction energy networks". In: *Proceedings of the International Conference on Machine Learning*. 2016.

[BMB18] Enzo Busseti, W Moursi, and Stephen Boyd. "Solution Refinement at Regular Points of Conic Problems". In: *arXiv preprint arXiv:1811.02157* (2018).

[BMR00] Ernesto G Birgin, José Mario Martínez, and Marcos Raydan. "Nonmonotone spectral projected gradient methods on convex sets". In: *SIAM Journal on Optimization* 10.4 (2000), pp. 1196–1211.

[BN01] Ahron Ben-Tal and Arkadi Nemirovski. *Lectures on modern convex optimization: analysis, algorithms, and engineering applications*. Vol. 2. Siam, 2001.

[Boe+14] Joschika Boedecker, Jost Tobias Springenberg, Jan Wulfing, and Martin Riedmiller. "Approximate Real-Time Optimal Control Based on Sparse Gaussian Process Models". In: *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. 2014.

[Boy+11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends® in Machine Learning* 3.1 (2011), pp. 1–122.

[BP16] Jonathan T Barron and Ben Poole. "The fast bilateral solver". In: *European Conference on Computer Vision*. Springer. 2016, pp. 617–632.

[Bro+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "OpenAI Gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[Bro+17] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. "Geometric deep learning: going beyond euclidean data". In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42.

[BS13] J Frédéric Bonnans and Alexander Shapiro. *Perturbation analysis of optimization problems*. Springer Science & Business Media, 2013.

[BS18] Shane Barratt and Rishi Sharma. "Optimizing for Generalization in Machine Learning with Cross-Validation Gradients". In: *arXiv preprint arXiv:1805.07072* (2018).

[BSS13] Philémon Brakel, Dirk Stroobandt, and Benjamin Schrauwen. "Training energy-based models for time-series imputation." In: *Journal of Machine Learning Research* 14.1 (2013), pp. 2771–2797.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[BYM17] David Belanger, Bishan Yang, and Andrew McCallum. "End-to-End Learning for Structured Prediction Energy Networks". In: *Proceedings of the International Conference on Machine Learning*. 2017.

[Che+15a] Liang-Chieh Chen, Alexander G Schwing, Alan L Yuille, and Raquel Urtasun. "Learning deep structured models". In: *Proceedings of the International Conference on Machine Learning*. 2015.

[Che+15b] Zhuo Chen, Lu Jiang, Wenlu Hu, Kiryong Ha, Brandon Amos, Padmanabhan Pillai, Alex Hauptmann, and Mahadev Satyanarayanan. "Early Implementation Experience with Wearable Cognitive Assistance Applications". In: *WearSys*. 2015.

[Che+17a] Yevgen Chebotar, Karol Hausman, Marvin Zhang, Gaurav Sukhatme, Stefan Schaal, and Sergey Levine. "Combining Model-Based and Model-Free Updates for Trajectory-Centric Reinforcement Learning". In: *arXiv preprint arXiv:1703.03078* (2017).

[Che+17b] Zhuo Chen et al. "An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM. 2017, p. 12.

[Che+18] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. "Neural ordinary differential equations". In: *Advances in Neural Information Processing Systems*. 2018, pp. 6572–6583.

[Cla75] Frank H Clarke. "Generalized gradients and applications". In: *Transactions of the American Mathematical Society* 205 (1975), pp. 247–262.

[DAK17] Priya L Donti, Brandon Amos, and J. Zico Kolter. "Task-based End-to-end Model Learning". In: *NIPS*. 2017.

[Dal+18] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. "Safe exploration in continuous action spaces". In: *arXiv preprint arXiv:1801.08757* (2018).

[Dav+16] Nigel Andrew Justin Davies, Nina Taft, Mahadev Satyanarayanan, Sarah Clinch, and Brandon Amos. "Privacy mediators: helping IoT cross the chasm". In: *HotMobile*. 2016.

[DB16] Steven Diamond and Stephen Boyd. "CVXPY: A Python-embedded modeling language for convex optimization". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 2909–2913.

[Dev+17] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. "Robustfill: Neural program learning under noisy I/O". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 990–998.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[Dia+17] Steven Diamond, Vincent Sitzmann, Felix Heide, and Gordon Wetzstein. "Unrolled optimization with deep priors". In: *arXiv preprint arXiv:1705.08041* (2017).

[Din77] U Dini. "Analisi infinitesimale, Lezioni dettate nella R". In: *Universitá di Pisa (1877/78)* (1877).

[DK17] Josip Djolonga and Andreas Krause. "Differentiable learning of submodular models". In: *Advances in Neural Information Processing Systems*. 2017, pp. 1013–1023.

[Dom12] Justin Domke. "Generic Methods for Optimization-Based Modeling." In: *AISTATS*. Vol. 22. 2012, pp. 318–326.

[DR09] Asen L Dontchev and R Tyrrell Rockafellar. "Implicit functions and solution mappings". In: *Springer Monogr. Math.* (2009).

[DR11] Marc Deisenroth and Carl E Rasmussen. "PILCO: A model-based and data-efficient approach to policy search". In: *Proceedings of the 28th International Conference on machine learning (ICML-11)*. 2011, pp. 465–472.

[Duc+08] John Duchi, Shai Shalev-Shwartz, Yoram Singer, and Tushar Chandra. "Efficient projections onto the l 1-ball for learning in high dimensions". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 272–279.

[FAL17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1126–1135.

[Far+17] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. "TreeQN and ATreeC: Differentiable Tree Planning for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1710.11417* (2017).

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. 10. Springer series in statistics New York, NY, USA: 2001.

[FI90] Anthony V Fiacco and Yo Ishizuka. "Sensitivity and stability analysis for nonlinear programming". In: *Annals of Operations Research* 27.1 (1990), pp. 215–235.

[FLA16] Chelsea Finn, Sergey Levine, and Pieter Abbeel. "Guided cost learning: Deep inverse optimal control via policy optimization". In: *International Conference on Machine Learning*. 2016, pp. 49–58.

[Foe+18] Jakob Foerster, Richard Y Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. "Learning with opponent-learning awareness". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2018, pp. 122–130.

[FP03] David A Forsyth and Jean Ponce. "A modern approach". In: *Computer vision: a modern approach* (2003), pp. 88–101.

[Gao+15] Ying Gao, Wenlu Hu, Kiryong Ha, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. *Are Cloudlets Necessary?* Tech. rep. Technical Report CMU-CS-15-139, CMU School of Computer Science, 2015.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[GBY06] Michael Grant, Stephen Boyd, and Yinyu Ye. "Disciplined convex programming". In: *Global optimization*. Springer, 2006, pp. 155–210.

[Gil+17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. "Neural message passing for quantum chemistry". In: *arXiv preprint arXiv:1704.01212* (2017).

[Goo+13] Ian Goodfellow, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. "Multi-prediction deep Boltzmann machines". In: *Advances in Neural Information Processing Systems*. 2013, pp. 548–556.

[Goo+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680.

[Goo+16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[Gou+16] Stephen Gould, Basura Fernando, Anoop Cherian, Peter Anderson, Rodrigo Santa Cruz, and Edison Guo. "On Differentiating Parameterized Argmin and Argmax Problems with Application to Bi-level Optimization". In: *arXiv preprint arXiv:1607.05447* (2016).

[Goy+18] Kartik Goyal, Graham Neubig, Chris Dyer, and Taylor Berg-Kirkpatrick. "A continuous relaxation of beam search for end-to-end training of neural sequence models". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[Gra+16] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 538.7626 (2016), p. 471.

[Gra+18] Will Grathwohl, Ricky TQ Chen, Jesse Betterncourt, Ilya Sutskever, and David Duvenaud. "Ffjord: Free-form continuous dynamics for scalable reversible generative models". In: *arXiv preprint arXiv:1810.01367* (2018).

[Gu+16a] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine. "Q-prop: Sample-efficient policy gradient with an off-policy critic". In: *arXiv preprint arXiv:1611.02247* (2016).

[Gu+16b] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. "Continuous Deep Q-Learning with Model-based Acceleration". In: *Proceedings of the International Conference on Machine Learning*. 2016.

[Gul+18] Caglar Gulcehre, Misha Denil, Mateusz Malinowski, Ali Razavi, Razvan Pascanu, Karl Moritz Hermann, Peter Battaglia, Victor Bapst, David Raposo, Adam Santoro, et al. "Hyperbolic attention networks". In: *arXiv preprint arXiv:1805.09786* (2018).

[GW08] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[GWD14] Alex Graves, Greg Wayne, and Ivo Danihelka. "Neural turing machines". In: *arXiv preprint arXiv:1410.5401* (2014).

[Ha+17] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. "You can teach elephants to dance: agile VM handoff for edge computing". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM. 2017, p. 12.

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *arXiv preprint arXiv:1512.03385* (2015).

[Hee+15] Nicolas Heess, Gregory Wayne, David Silver, Tim Lillicrap, Tom Erez, and Yuval Tassa. "Learning continuous control policies by stochastic value gradients". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2944–2952.

[Hen+18] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. "Deep reinforcement learning that matters". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[Hil+15] Felix Hill, Antoine Bordes, Sumit Chopra, and Jason Weston. "The goldilocks principle: Reading children's books with explicit memory representations". In: *arXiv preprint arXiv:1511.02301* (2015).

[HLW16] Gao Huang, Zhuang Liu, and Kilian Q Weinberger. "Densely Connected Convolutional Networks". In: *arXiv preprint arXiv:1608.06993* (2016).

[HSF18] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. "Matrix capsules with EM routing". In: (2018).

[Hu+15] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. "The Case for Offload Shaping". In: *HotMobile*. 2015.

[Hu+16] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. "Quantifying the impact of edge computing on mobile applications". In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM. 2016, p. 5.

[HYL17] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *Advances in Neural Information Processing Systems*. 2017, pp. 1024–1034.

[Ing+18] John Ingraham, Adam Riesselman, Chris Sander, and Debora Marks. "Learning Protein Structure with a Differentiable Simulator". In: (2018).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of The 32nd International Conference on Machine Learning*. 2015, pp. 448–456.

[Joh+16] Matthew Johnson, David K Duvenaud, Alex Wiltschko, Ryan P Adams, and Sandeep R Datta. "Composing graphical models with neural networks for structured representations and fast inference". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2946–2954.

[Jor15] Christopher Jordan-Squire. "Convex Optimization over Probability Measures". PhD thesis. 2015.

[JRB18] Rico Jonschkowski, Divyam Rastogi, and Oliver Brock. "Differentiable particle filters: End-to-end learning with algorithmic priors". In: *arXiv preprint arXiv:1805.11122* (2018).

[KB14] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KC88] Michael Peter Kennedy and Leon O Chua. "Neural networks for nonlinear programming". In: *IEEE Transactions on Circuits and Systems* 35.5 (1988), pp. 554–562.

[KF09] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[KHL17] Peter Karkus, David Hsu, and Wee Sun Lee. "Qmdp-net: Deep learning for planning under partial observability". In: *Advances in Neural Information Processing Systems*. 2017, pp. 4697–4707.

[KP13] Karl Kunisch and Thomas Pock. "A bilevel optimization approach for parameter learning in variational models". In: *SIAM Journal on Imaging Sciences* 6.2 (2013), pp. 938–983.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[KTV08] Ioannis Katakis, Grigorios Tsoumakas, and Ioannis Vlahavas. "Multilabel text classification for automated tag suggestion". In: *ECML PKDD discovery challenge* 75 (2008).

[Kur+18] Thanard Kurutach, Aviv Tamar, Ge Yang, Stuart J Russell, and Pieter Abbeel. "Learning plannable representations with causal infogan". In: *Advances in Neural Information Processing Systems*. 2018, pp. 8733–8744.

[KW16] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).

[LA14] Sergey Levine and Pieter Abbeel. "Learning neural network policies with guided policy search under unknown dynamics". In: *Advances in Neural Information Processing Systems*. 2014, pp. 1071–1079.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher JC Burges. *The MNIST database of handwritten digits*. 1998.

[LeC+06] Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and F Huang. "A tutorial on energy-based learning". In: *Predicting structured data* 1 (2006).

[Lee+19] Kwonjoon Lee, Subhransu Maji, Avinash Ravichandran, and Stefano Soatto. "Meta-Learning with Differentiable Convex Optimization". In: *arXiv preprint arXiv:1904.03758* (2019).

[Lev+16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. "End-to-end training of deep visuomotor policies". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1334–1373.

[Lev17a] Sergey Levine. *Introduction to Reinforcement Learning*. Berkeley CS 294-112: Deep Reinforcement Learning. 2017.

[Lev17b] Sergey Levine. *Optimal Control and Planning*. Berkeley CS 294-112: Deep Reinforcement Learning. 2017.

[LFK18] Chun Kai Ling, Fei Fang, and J. Zico Kolter. "What game are we playing? end-to-end learning in normal and extensive form games". In: *arXiv preprint arXiv:1805.02777* (2018).

[Li+18] Xiang Li, Luke Vilnis, Dongxu Zhang, Michael Boratko, and Andrew McCallum. "Smoothing the Geometry of Probabilistic Box Embeddings". In: (2018).

[Li94] Stan Z Li. "Markov random field models in computer vision". In: *European conference on computer vision*. Springer. 1994, pp. 361–370.

[Lil+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[Lil+93] Walter E Lillo, Mei Heng Loh, Stefen Hui, and Stanislaw H Zak. "On solving constrained optimization problems with neural networks: A penalty method approach". In: *IEEE Transactions on neural networks* 4.6 (1993), pp. 931–940.

[LKS15] Ian Lenz, Ross A Knepper, and Ashutosh Saxena. "DeepMPC: Learning Deep Latent Features for Model Predictive Control." In: *Robotics: Science and Systems*. 2015.

[LMP01] John Lafferty, Andrew McCallum, and Fernando CN Pereira. "Conditional random fields: Probabilistic models for segmenting and labeling sequence data". In: (2001).

[Lob+98] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. "Applications of second-order cone programming". In: *Linear algebra and its applications* 284.1-3 (1998), pp. 193–228.

[Löt84] Per Lötstedt. "Numerical simulation of time-dependent contact and friction problems in rigid body mechanics". In: *SIAM journal on scientific and statistical computing* 5.2 (1984), pp. 370–393.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[LT04] Weiwei Li and Emanuel Todorov. "Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems". In: 2004.

[MA16] Andre Martins and Ramon Astudillo. "From softmax to sparsemax: A sparse model of attention and multi-label classification". In: *International Conference on Machine Learning*. 2016, pp. 1614–1623.

[MB09] Alessandro Magnani and Stephen P Boyd. "Convex piecewise-linear fitting". In: *Optimization and Engineering* 10.1 (2009), pp. 1–17.

[MB12] Jacob Mattingley and Stephen Boyd. "CVXGEN: A code generator for embedded convex optimization". In: *Optimization and Engineering* 13.1 (2012), pp. 1–27.

[MB18] Arthur Mensch and Mathieu Blondel. "Differentiable dynamic programming for structured prediction and attention". In: *arXiv preprint arXiv:1802.03676* (2018).

[MBP12] Julien Mairal, Francis Bach, and Jean Ponce. "Task-driven dictionary learning". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34.4 (2012), pp. 791–804.

[Met+16] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. "Unrolled Generative Adversarial Networks". In: *arXiv preprint arXiv:1611.02163* (2016).

[ML99] Manfred Morari and Jay H Lee. "Model predictive control: past, present and future". In: *Computers & Chemical Engineering* 23.4 (1999), pp. 667–682.

[MN88] X Magnus and Heinz Neudecker. "Matrix differential calculus". In: *New York* (1988).

[Mni+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[Mni+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[Mni+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*. 2016, pp. 1928–1937.

[Mon+17] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. "Geometric deep learning on graphs and manifolds using mixture model cnns". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5115–5124.

[Mor61] Jean Jacques Moreau. "Décomposition orthogonale d'un espace hilbertien selon deux cônes mutuellement polaires". In: *Comptes rendus hebdomadaires des séances de l'Académie des sciences* 255 (1961), pp. 238–240.

[Nag+17] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning". In: *arXiv preprint arXiv:1708.02596*. 2017.

[NB17] Vlad Niculae and Mathieu Blondel. "A regularized framework for sparse and structured neural attention". In: *Advances in Neural Information Processing Systems*. 2017, pp. 3338–3348.

[NH10] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 807–814.

[Nic+18] Vlad Niculae, André FT Martins, Mathieu Blondel, and Claire Cardie. "SparseMAP: Differentiable sparse structured inference". In: *arXiv preprint arXiv:1802.04223* (2018).

[NLS15] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. "Neural programmer: Inducing latent programs with gradient descent". In: *arXiv preprint arXiv:1511.04834* (2015).

[NR+00] Andrew Y Ng, Stuart J Russell, et al. "Algorithms for inverse reinforcement learning." In: *Icml*. Vol. 1. 2000, p. 2.

[NW06] Jorge Nocedal and Stephen J Wright. *Sequential quadratic programming*. Springer, 2006.

[ODo+16] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. "Conic optimization via operator splitting and homogeneous self-dual embedding". In: *Journal of Optimization Theory and Applications* 169.3 (2016), pp. 1042–1068.

[Oh+16] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. "Control of Memory, Active Perception, and Action in Minecraft". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)* (2016).

[Oli06] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[ORA17] Masashi Okada, Luca Rigazio, and Takenobu Aoshima. "Path Integral Networks: End-to-End Differentiable Optimal Control". In: *arXiv preprint arXiv:1706.09597* (2017).

[OSL17] Junhyuk Oh, Satinder Singh, and Honglak Lee. "Value prediction network". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6120–6130.

[Par+16] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. "Neuro-symbolic program synthesis". In: *arXiv preprint arXiv:1611.01855* (2016).

[Pas+17a] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. "Learning model-based planning from scratch". In: *arXiv preprint arXiv:1707.06170* (2017).

[Pas+17b] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". In: *NIPS Autodiff Workshop* (2017).

[Pat+18] Deepak Pathak, Parsa Mahmoudieh, Guanghao Luo, Pulkit Agrawal, Dian Chen, Yide Shentu, Evan Shelhamer, Jitendra Malik, Alexei A Efros, and

Trevor Darrell. "Zero-shot visual imitation". In: *arXiv preprint arXiv:1804.08606* (2018).

[PBX09] Jian Peng, Liefeng Bo, and Jinbo Xu. "Conditional neural fields". In: *Advances in neural information processing systems*. 2009, pp. 1419–1427.

[PD11] Hoifung Poon and Pedro Domingos. "Sum-product networks: A new deep architecture". In: *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*. 2011, pp. 337–346.

[PDT18] Tu-Hoa Pham, Giovanni De Magistris, and Ryuki Tachibana. "Optlayer-practical constrained optimization for deep reinforcement learning in the real world". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 6236–6243.

[Ped+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine learning in Python". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Per+18] Marcus Pereira, David D. Fan, Gabriel Nakajima An, and Evangelos Theodorou. "MPC-Inspired Neural Network Policies for Sequential Decision Making". In: *arXiv preprint arXiv:1802.05803* (2018).

[Pol64] Boris T Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[Pon+18] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. "Temporal Difference Models: Model-Free Deep RL for Model-Based Control". In: *arXiv preprint arXiv:1802.09081* (2018).

[PS17] Emilio Parisotto and Ruslan Salakhutdinov. "Neural map: Structured memory for deep reinforcement learning". In: *arXiv preprint arXiv:1702.08360* (2017).

[PS82] Christopher C Paige and Michael A Saunders. "LSQR: An algorithm for sparse linear equations and sparse least squares". In: *ACM Transactions on Mathematical Software (TOMS)* 8.1 (1982), pp. 43–71.

[RBZ07] Nathan D Ratliff, J Andrew Bagnell, and Martin Zinkevich. "(Approximate) Subgradient Methods for Structured Prediction". In: *International Conference on Artificial Intelligence and Statistics*. 2007, pp. 380–387.

[RD15] Scott Reed and Nando De Freitas. "Neural programmer-interpreters". In: *arXiv preprint arXiv:1511.06279* (2015).

[RHW88] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (1988), p. 1.

[RL16] Sachin Ravi and Hugo Larochelle. "Optimization as a model for few-shot learning". In: (2016).

[Roc70] R Tyrrell Rockafellar. "Convex Analysis Princeton University Press". In: *Princeton, NJ* (1970).

[San+17] Adam Santoro, David Raposo, David GT Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. "A simple neural network module for relational reasoning". In: *arXiv preprint arXiv:1706.01427* (2017).

[Sat+15] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. "Edge Analytics in the Internet of Things". In: *IEEE Pervasive Computing* 2 (2015), pp. 24–31.

[SB+98] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[SB11] Shankar Sastry and Marc Bodson. *Adaptive control: stability, convergence and robustness*. Courier Corporation, 2011.

[Sch+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust region policy optimization". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, pp. 1889–1897.

[Sch+16] John Schulman, Philpp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *International Conference on Learning Representations* (2016).

[Sch15] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[Sch97] Jeff G Schneider. "Exploiting model uncertainty estimates for safe dynamic control learning". In: *Advances in neural information processing systems*. 1997, pp. 1047–1053.

[SFH17] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. "Dynamic routing between capsules". In: *Advances in neural information processing systems*. 2017, pp. 3856–3866.

[SH94] Ferdinando S Samaria and Andy C Harter. "Parameterisation of a stochastic model for human face identification". In: *Applications of Computer Vision, 1994., Proceedings of the Second IEEE Workshop on*. IEEE. 1994, pp. 138–142.

[She+18] Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. "Ordered Neurons: Integrating Tree Structures into Recurrent Neural Networks". In: *arXiv preprint arXiv:1810.09536* (2018).

[Sil+16] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, et al. "The predictron: End-to-end learning and planning". In: *arXiv preprint arXiv:1612.08810* (2016).

[SL91] Patrice Simard and Yann LeCun. "Reverse TDNN: an architecture for trajectory generation". In: *Advances in Neural Information Processing Systems*. Citeseer. 1991, pp. 579–588.

[SM+12] Charles Sutton, Andrew McCallum, et al. "An introduction to conditional random fields". In: *Foundations and Trends® in Machine Learning* 4.4 (2012), pp. 267–373.

[SNW12] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

[SR14] Uwe Schmidt and Stefan Roth. "Shrinkage fields for effective image restoration". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 2774–2781.

[SRE11] Veselin Stoyanov, Alexander Ropson, and Jason Eisner. "Empirical Risk Minimization of Graphical Model Parameters Given Approximate Inference, Decoding, and Model Structure." In: *AISTATS*. 2011, pp. 725–733.

[Sri+18] Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. "Universal Planning Networks". In: *arXiv preprint arXiv:1804.00645* (2018).

[Sun+17] Liting Sun, Cheng Peng, Wei Zhan, and Masayoshi Tomizuka. "A Fast Integrated Planning and Control Framework for Autonomous Driving via Imitation Learning". In: *arXiv preprint arXiv:1707.02515*. 2017.

[Sut90] Richard S Sutton. "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming". In: *Proceedings of the seventh international conference on machine learning*. 1990, pp. 216–224.

[SVL08] Alex J. Smola, S.v.n. Vishwanathan, and Quoc V. Le. "Bundle Methods for Machine Learning". In: *Advances in Neural Information Processing Systems 20*. Ed. by J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis. Curran Associates, Inc., 2008, pp. 1377–1384.

[SWF+15] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. "End-to-end memory networks". In: *Advances in neural information processing systems*. 2015, pp. 2440–2448.

[SZ14] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[Sze+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[Sze10] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[Tam+16] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. "Value iteration networks". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2154–2162.

[Tam+17] Aviv Tamar, Garrett Thomas, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. "Learning from the hindsight plan—Episodic MPC improvement". In: *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE. 2017, pp. 336–343.

[Tap+07] Marshall F Tappen, Ce Liu, Edward H Adelson, and William T Freeman. "Learning gaussian conditional random fields for low-level vision". In: *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE. 2007, pp. 1–8.

[Tas+05] Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. "Learning structured prediction models: A large margin approach". In: *Proceedings of the 22nd International Conference on Machine Learning*. ACM. 2005, pp. 896–903.

[TBS10] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. "A generalized path integral control approach to reinforcement learning". In: *Journal of Machine Learning Research* 11.Nov (2010), pp. 3137–3181.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033.

[TGK04] Ben Taskar, Carlos Guestrin, and Daphne Koller. "Max-margin Markov networks". In: *Advances in neural information processing systems*. 2004, pp. 25–32.

[TMT14] Yuval Tassa, Nicolas Mansard, and Emo Todorov. "Control-limited differential dynamic programming". In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE. 2014, pp. 1168–1175.

[TSK18] Sebastian Tschiatschek, Aytunc Sahin, and Andreas Krause. "Differentiable submodular maximization". In: *arXiv preprint arXiv:1803.01785* (2018).

[Tso+05] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. "Large margin methods for structured and interdependent output variables". In: *Journal of Machine Learning Research* 6 (2005), pp. 1453–1484.

[Tso+11] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. "Mulan: A java library for multi-label learning". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2411–2414.

[TT18] Chengzhou Tang and Ping Tan. "Ba-net: Dense bundle adjustment network". In: *arXiv preprint arXiv:1806.04807* (2018).

[UVL18] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. "Deep image prior". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9446–9454.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in Neural Information Processing Systems*. 2017, pp. 5998–6008.

[VD95] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[Ven+16] Arun Venkatraman, Roberto Capobianco, Lerrel Pinto, Martial Hebert, Daniele Nardi, and J Andrew Bagnell. "Improved learning of dynamics models for control". In: *International Symposium on Experimental Robotics*. Springer. 2016, pp. 703–713.

[Wan+17] Junjue Wang, Brandon Amos, Anupam Das, Padmanabhan Pillai, Norman Sadeh, and Mahadev Satyanarayanan. "A Scalable and Privacy-Aware IoT

Service for Live Video Analytics". In: *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM. 2017, pp. 38–49.

[Wan+18] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. "Non-local neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7794–7803.

[Was13] Larry Wasserman. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013.

[Wat+15] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. "Embed to control: A locally linear latent dynamics model for control from raw images". In: *Advances in neural information processing systems*. 2015, pp. 2746–2754.

[WAT17] Grady Williams, Andrew Aldrich, and Evangelos A Theodorou. "Model predictive path integral control: From theory to parallel computation". In: *Journal of Guidance, Control, and Dynamics* 40.2 (2017), pp. 344–357.

[Web+17] Théophane Weber, Sébastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. "Imagination-Augmented Agents for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1707.06203* (2017).

[WFU16] Shenlong Wang, Sanja Fidler, and Raquel Urtasun. "Proximal deep structured models". In: *Advances in Neural Information Processing Systems*. 2016, pp. 865–873.

[Wil+16] Grady Williams, Paul Drews, Brian Goldfain, James M Rehg, and Evangelos A Theodorou. "Aggressive driving with model predictive path integral control". In: *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1433–1440.

[Wri97] Stephen J Wright. *Primal-dual interior-point methods*. Siam, 1997.

[XC18] Zhang Xinyi and Lihui Chen. "Capsule Graph Neural Network". In: (2018).

[XLH17] Zhaoming Xie, C. Karen Liu, and Kris Hauser. "Differential Dynamic Programming with Nonlinear Constraints". In: *International Conference on Robotics and Automation (ICRA)*. 2017.

[XMS16] Caiming Xiong, Stephen Merity, and Richard Socher. "Dynamic memory networks for visual and textual question answering". In: *International conference on machine learning*. 2016, pp. 2397–2406.

[Xu+18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. "How Powerful are Graph Neural Networks?" In: *arXiv preprint arXiv:1810.00826* (2018).

[YK15] Fisher Yu and Vladlen Koltun. "Multi-scale context aggregation by dilated convolutions". In: *arXiv preprint arXiv:1511.07122* (2015).

[YTM94] Yinyu Ye, Michael J Todd, and Shinji Mizuno. "An $O(\sqrt{nL})$-iteration homogeneous and self-dual linear programming algorithm". In: *Mathematics of Operations Research* 19.1 (1994), pp. 53–67.

[Zah+17] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. "Deep sets". In: *Advances in neural information processing systems*. 2017, pp. 3391–3401.

[Zha+16] Han Zhao, Tameem Adel, Geoff Gordon, and Brandon Amos. "Collapsed Variational Inference for Sum-Product Networks". In: *ICML*. 2016.

[Zhe+15] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. "Conditional random fields as recurrent neural networks". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1529–1537.